

----- Begin Cut here for COMPONENT Declaration ----

-- COMP_TAG

COMPONENT fifo_generator_17_out

PORT (

rst : IN STD_LOGIC;

wr_clk : IN STD_LOGIC;

rd_clk : IN STD_LOGIC;

din : IN STD_LOGIC_VECTOR(16 DOWNT0 0);

wr_en : IN STD_LOGIC;

rd_en : IN STD_LOGIC;

dout : OUT STD_LOGIC_VECTOR(16 DOWNT0 0);

full : OUT STD_LOGIC;

empty : OUT STD_LOGIC;

wr_rst_busy : OUT STD_LOGIC;

rd_rst_busy : OUT STD_LOGIC

);

END COMPONENT;

-- COMP_TAG_END ----- End COMPONENT Declaration ----

-- The following code must appear in the VHDL
architecture

-- body. Substitute your own instance name and net
names.

----- Begin Cut here for INSTANTIATION Template ----

- INST_TAG

your_instance_name : fifo_generator_17_out

PORT MAP (

rst => rst,

wr_clk => wr_clk,

rd_clk => rd_clk,

din => din,

wr_en => wr_en,

rd_en => rd_en,

dout => dout,

full => full,

empty => empty,

wr_rst_busy => wr_rst_busy,

```
rd_rst_busy => rd_rst_busy  
);
```

how to create an instance of this IP?

Good question — let's go step by step.


You've pasted the **component declaration** and **instantiation template** for a Xilinx FIFO IP (`fifo_generator_17_out`).

To create and use an instance of this IP in your **VHDL design**, you must go through two stages:

1 Make sure the FIFO IP exists in your project

This component corresponds to a **FIFO Generator** IP created using Vivado's IP catalog.

► Steps in Vivado:

1. **Open Vivado** → **IP Catalog**.
2. Search for **FIFO Generator**.
3. Double-click it → Configure it (width, depth, clocks, etc.).
4. **Set the component name** to match the one in your code:
 `fifo_generator_17_out`
(you can change it if you want, but make sure the name in VHDL matches exactly).
5. Click **"Generate Output Products"** → **"Synthesis and Simulation"**.

Vivado will generate:

 Copy code

```
fifo_generator_17_out.vhd  
fifo_generator_17_out.xci  
fifo_generator_17_out.dcp
```

2 Instantiate it in your design

Once the IP is generated and added to your project, you can instantiate it anywhere inside an architecture (e.g. inside your top-level design).

Here's a **concrete example**:

vhdl

 Copy code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity top_module is
  Port (
    rst      : in  std_logic;
    wr_clk    : in  std_logic;
    rd_clk    : in  std_logic;
    din       : in  std_logic_vector(16 downto 0);
    wr_en     : in  std_logic;
    rd_en     : in  std_logic;
    dout      : out std_logic_vector(16 downto 0);
    full      : out std_logic;
    empty     : out std_logic
  );
end top_module;

architecture Behavioral of top_module is

  -- Component declaration (copied from Vivado template)
  COMPONENT fifo_generator_17_out
  PORT (
    rst      : IN  STD_LOGIC;
    wr_clk    : IN  STD_LOGIC;
    rd_clk    : IN  STD_LOGIC;
    din       : IN  STD_LOGIC_VECTOR(16 DOWNT0 0);
    wr_en     : IN  STD_LOGIC;
    rd_en     : IN  STD_LOGIC;
    dout      : OUT STD_LOGIC_VECTOR(16 DOWNT0 0);
    full      : OUT STD_LOGIC;
    empty     : OUT STD_LOGIC;
    wr_rst_busy : OUT STD_LOGIC;
    rd_rst_busy : OUT STD_LOGIC
```

```

    );
END COMPONENT;

-- Internal signals (optional)
signal wr_rst_busy_s, rd_rst_busy_s : std_logic;

begin

-- FIFO instance
fifo_inst : fifo_generator_17_out
PORT MAP (
    rst          => rst,
    wr_clk       => wr_clk,
    rd_clk       => rd_clk,
    din          => din,
    wr_en        => wr_en,
    rd_en        => rd_en,
    dout         => dout,
    full         => full,
    empty        => empty,
    wr_rst_busy  => wr_rst_busy_s,
    rd_rst_busy  => rd_rst_busy_s
);

end Behavioral;

```

3 Add the IP to your synthesis/simulation sources

In **Vivado**, make sure the generated IP (`fifo_generator_17_out.xci`) is included in:

- **Project** → **Sources** → **Design Sources**
- If you use simulation: also under **Simulation Sources**

Vivado will automatically handle compilation and linking.

4 Optional: Use `work.fifo_generator_17_out` instead of **COMPONENT**

If you have VHDL-2008 enabled and Vivado supports it, you can directly **instantiate by entity name** (without component declaration):

vhdl

 Copy code

```
fifo_inst : entity work.fifo_generator_17_out
  port map (
    rst => rst,
    wr_clk => wr_clk,
    rd_clk => rd_clk,
    din => din,
    wr_en => wr_en,
    rd_en => rd_en,
    dout => dout,
    full => full,
    empty => empty,
    wr_rst_busy => wr_rst_busy_s,
    rd_rst_busy => rd_rst_busy_s
  );
```

This is cleaner and avoids copying the component declaration manually.

Would you like me to show you **how to connect this FIFO between a producer and consumer module** (for example, between two clock domains)? It helps to see how `wr_en`, `rd_en`, `full`, and `empty` interact in practice.

     ...

```
[Vivado 12-3630] The destination file
'/home/UFAD/audreycyriell.mo/Documents/reconfig1/la
b5-lab-5-group-15/ip_repo/fifo_1.0/src/fifo17.vhd'
already exists, please use -force if you want to overwrite!
```

That Vivado message:

swift

 Copy code

[Vivado 12-3630] The destination file '/home/UFAD/audreycyriell.mo/Documents'

is simply telling you that Vivado is trying to generate or copy an IP source file (fifo17.vhd) into your repository folder, but a file with the same name **already exists**, so it refuses to overwrite it by default.

🚫 Here's what it means and how to fix it

You have two main options:

✅ Option 1: Force overwrite the file

If you're regenerating the IP intentionally (e.g., reconfigured FIFO, changed parameters), you can safely overwrite the existing source.

► Using the Vivado GUI

- When Vivado prompts during IP generation, **check the box "Overwrite existing files"** or **click OK** when it warns you.
- Alternatively, delete the old IP folder before regenerating:

```
bash
```

[📄 Copy code](#)

```
rm -rf /home/UFAD/audreycyriell.mo/Documents/reconfig1/lab5-lab-5-group
```

Then regenerate the IP normally.

► Using the Tcl console

If you're running Vivado commands manually or through a script:

```
tcl
```

[📄 Copy code](#)

```
write_vhdl -force /home/UFAD/audreycyriell.mo/Documents/reconfig1/lab5-lab
```

or when regenerating the IP:

```
tcl
```

```
generate_target all [get_ips fifo_generator_17_out] -force
```

That `-force` flag overwrites existing files.

✅ Option 2: Keep the existing file

If you didn't mean to regenerate the IP, you can just **ignore the message** — your FIFO IP is already generated and available.

You can verify this by checking that:

```
pgsql
```

[Copy code](#)

```
fifo17.vhd
```

```
fifo_generator_17_out.xci
```

```
component.xml
```

exist inside:

```
bash
```

[Copy code](#)

```
ip_repo/fifo_1.0/src/
```

If these are present, your IP is already built and ready to use.

🧹 Optional: clean and rebuild IPs

If Vivado keeps giving duplicate or stale file warnings:

```
bash
```

[Copy code](#)

```
cd /home/UFAD/audreycyriell.mo/Documents/reconfig1/lab5-lab-5-group-15/
```

```
rm -rf ip_repo/
```

Then in Vivado:

- Recreate the IP Repository via **Project Settings** → **IP** → **Repository** → **Add Repository**.

- Regenerate the FIFO.

Would you like me to show you the **exact Tcl commands** to regenerate the FIFO IP cleanly with `-force` inside your project?



```
WARNING: [IP_Flow 19-395] Problem validating against
XML schema: : Unexpected end of message
ERROR: [Vivado 12-3629] The destination directory
'/home/UFAD/audreycyriell.mo/Documents/reconfig1/final_project/provided_code/dram_test/dram_test_1.0/src/dma_fifo' already exists, please use -force if you want to overwrite!
```

These two messages mean **Vivado tried to regenerate an IP**, but the IP folder already exists and contains files that Vivado cannot overwrite unless you force it.

Let me break down the two warnings/errors and show you exactly how to fix them cleanly.

✓ Understanding the Messages

1 WARNING: [IP_Flow 19-395] Problem validating against XML schema

This happens when Vivado reads an IP folder whose metadata (`component.xml` , `.xci`) is:

- corrupted
- partially generated
- incomplete
- out-of-sync with the Vivado version

It usually happens when:

- You copied an IP folder manually between projects
- Regeneration failed halfway
- Vivado was closed before the IP finished generating

Conclusion: the IP repository is in a broken state.

2 ERROR: [Vivado 12-3629] The destination directory already exists

swift

 Copy code

```
'/home/UFAD/.../dram_test/dram_test_1.0/src/dma_fifo' already exists
```

Vivado wants to regenerate the IP at that path, but cannot continue because it won't overwrite the directory unless you use **-force**.

✓ FIX OPTIONS (CHOOSE ONE)

✓ Option A — Safest: Delete the broken IP directory, then regenerate

This is the cleanest and most reliable fix.

Step 1 — Delete the existing IP directory

bash

 Copy code

```
rm -rf /home/UFAD/audreycyriell.mo/Documents/reconfig1/final_project/provi
```

(Be careful: `rm -rf` removes everything inside the folder.)

Step 2 — In Vivado, regenerate IP:

Right-click the IP → Generate Output Products

or

CSS

 Copy code

```
generate_target all [get_ips dma_fifo]
```

This fixes both the XML warning and the overwrite error.

✓ Option B — Force Vivado to overwrite the directory

Run in Tcl Console:

```
tcl
```

[Copy code](#)

```
generate_target all [get_ips dma_fifo] -force
```

Or for all IPs:

```
tcl
```

[Copy code](#)

```
upgrade_ip [get_ips]  
generate_target all [get_ips] -force
```

This will **overwrite XML, VHDL, and metadata files** and rebuild the IP completely.

✓ Option C — Remove the IP from the project and re-add it

If the repository structure is too messy, do this:

Step 1:

Vivado → **Sources Panel** → *IP Sources*

Right-click the faulty IP → **Remove**.

Step 2:

Add the IP back via:

- **Project Settings** → **IP** → **Repository** → **Add Repository**
- Point to:

```
/final_project/provided_code/dram_test/dram_test_1.0
```

Step 3:

Regenerate the IP.

★ Recommended Approach

Use Option A: delete the broken folder, then regenerate.

It guarantees no stale XML or corrupted IP metadata remains.

If you want, I can:

- ✓ check your exact project structure
- ✓ tell you the correct IP names (from your `.xci`)
- ✓ give you the exact Tcl command to regenerate all IPs cleanly




Just send me the output of:

```
bash
```

 Copy code

```
ls /home/UFAD/audreycyriell.mo/Documents/reconfig1/final_project/provided_
```



```
generate_target all [get_ips dma_fifo]  
WARNING: [Coretcl 2-176] No IPs found  
generate_target all [get_ips dma_fifo] -force  
WARNING: [Coretcl 2-176] No IPs found
```

The message **“No IPs found”** means that **Vivado does NOT recognize** `dma_fifo` **as an IP in your project** — even though the folder exists on disk.

This happens when:

- The IP was **copied manually** into the project, but never *added* to Vivado
- The `.xci` file is missing or corrupted

- The IP repository is not registered in **Project Settings** → **IP** → **Repository**
- Vivado is looking in the wrong place

Let me walk you through fixing this correctly.

✓ Step 1 — Check if the IP has a `.xci` file

Please run this command in a terminal:

```
bash
```

[Copy code](#)

```
ls /home/UFAD/audreycyriell.mo/Documents/reconfig1/final_project/provided_
```

You are looking for something like:

```
pgsql
```

[Copy code](#)

```
dma_fifo.xci  
component.xml
```

If `.xci` **is missing**, the IP is **not a valid Vivado IP**, and Vivado will never detect it.

Please send me the output of that command so I can guide you.

✓ Step 2 — Check what IPs Vivado sees

In the Vivado Tcl Console, run:

```
tcl
```

[Copy code](#)

```
get_ips
```

This will list **all IPs** Vivado knows about.

If `dma_fifo` is not in the list → it is NOT registered as an IP.

If the IP does not show up:

Then you must *re-add the IP repository*:

1 Go to

Tools → Project Settings → IP → Repository

2 Click "Add Repository"

Add this folder:

```
swift
```

[Copy code](#)

```
/home/UFAD/audreycyriell.mo/Documents/reconfig1/final_project/provided_code
```

After adding, Vivado should detect IPs in that directory.

You will then see something like:

```
scss
```

[Copy code](#)

```
dma_fifo (1.0)
```

If the repository STILL doesn't show the IP

Then the IP folder is **corrupted** (missing XML or XCI).

In that case, you must:

1 Delete the damaged IP directory

```
bash
```

[Copy code](#)

```
rm -rf /home/UFAD/.../dram_test_1.0/src/dma_fifo
```

2 Recreate the IP manually in Vivado:

IP Catalog → FIFO Generator → create again → name it dma_fifo

Generate Output Products

Vivado will produce:

```
pgsql
```

[Copy code](#)

```
dma_fifo.xci  
dma_fifo.vhd  
component.xml
```

Then it will work.

★ To proceed

Please send me the output of:

◆ 1.

```
bash
```

[Copy code](#)

```
ls /home/UFAD/audreycyriell.mo/Documents/reconfig1/final_project/provided_
```

◆ 2.

```
tcl
```

[Copy code](#)

```
get_ips
```

Once I see those, I can tell you the exact next step.

[Copy](#) [Like](#) [Reply](#) [Share](#) [Refresh](#) [More](#)

```
get_ips  
mmap_data_fifo mmap_request_fifo
```

Great — this tells us exactly what is happening.

```
nginx
```

[Copy code](#)

```
get_ips  
mmap_data_fifo  mmap_request_fifo
```

➔ **Vivado only knows about two IPs** in your project:

- `mmap_data_fifo`
- `mmap_request_fifo`

`dma_fifo` **is NOT an IP in your project.**

This is why:

```
pgsql
```

📄 Copy code

```
get_ips dma_fifo
```

```
→ WARNING: No IPs found
```

Even though there is a folder on disk named `dma_fifo`, **Vivado does not recognize it**, because it is either:

- missing its `.xci` file
- missing `component.xml`
- not in the registered repository
- or corrupted

Let's fix this cleanly.

✅ **FIX: Recreate the `dma_fifo` IP so Vivado recognizes it**

Since `dma_fifo` is *not* detected as an IP, the safest solution is to **delete the broken folder and recreate the IP**.

1 Delete the broken IP folder

Run this in a terminal:

```
bash
```

📄 Copy code

```
rm -rf /home/UFAD/audreycyriell.mo/Documents/reconfig1/final_project/provi
```

This cleans the corrupted IP directory.

2 Recreate the IP inside Vivado

Steps:

1. Open Vivado
2. Go to **IP Catalog**
3. Search for: **FIFO Generator**
4. Double-click to create a new FIFO
5. Set the **IP name** to:

```
nginx
```

 Copy code

```
dma_fifo
```

6. Configure FIFO the way you need:
 - Data width
 - Depth
 - Write clock / Read clock
7. Click **OK**
8. When Vivado asks to generate files → choose **Generate Output Products**

Vivado will create a valid IP directory:

```
pgsql
```

 Copy code

```
dma_fifo.xci  
component.xml  
dma_fifo.vhd
```

Inside this folder:

```
swift
```

 Copy code

```
/provided_code/dram_test/dram_test_1.0/src/dma_fifo/
```


3 Verify Vivado now sees the IP

In Tcl:

```
tcl
```

 Copy code

```
get_ips
```

You should now see:

```
nginx
```

 Copy code

```
mmap_data_fifo
```

```
mmap_request_fifo
```

```
dma_fifo
```

Now all IP commands will work:

```
tcl
```

 Copy code

```
generate_target all [get_ips dma_fifo]
```

```
generate_target all [get_ips] -force
```

★ If you want, I can also:

- ✓ Tell you the exact FIFO parameters needed for your lab
- ✓ Help you integrate `dma_fifo` into your design
- ✓ Check your IP repository structure for correctness

Just let me know.

     ...

[IP_Flow 19-8166] Failed to verify json document against the schema 'xilinx.com:schema:json_instance:1.0'.

```
{  
  "required": {  
    "missing": [ "gen_directory" ],
```

```
"instanceRef": "#/ip_inst",  
"schemaRef":  
"file:///json_instance_v1_0.json#/properties/ip_inst"  
}  
}
```

This error:

CSS

 Copy code

```
[IP_Flow 19-8166] Failed to verify json document against the schema ...  
missing: [ "gen_directory" ]
```

means:

Your IP's metadata (.json / .xci) is corrupted or incomplete.

Vivado reads a JSON structure inside the .xci file and expects a field called:

nginx

 Copy code

```
gen_directory
```

This tells Vivado where to write the generated HDL files.

If it's missing → the IP cannot load → you get this JSON schema failure.

This is **very common** when:

- An IP folder was copied manually
- The .xci file was partially deleted
- The IP was generated in an older version of Vivado and imported incorrectly
- A folder existed before generating a new IP with the same name
- Repository paths changed

How to fix it (guaranteed fix)

You must **remove the corrupted IP completely** and regenerate it cleanly.

1 Delete the broken IP folder

Run this in a terminal:

```
bash
```

[Copy code](#)

```
rm -rf /home/UFAD/audreycyriell.mo/Documents/reconfig1/final_project/provi
```

⚠ **IMPORTANT:** The folder *must be deleted*, not overwritten.

2 Remove the IP from Vivado (if it appears)

In Vivado:

- Open **Project Manager** → **Sources** → **IP Sources**
- If `dma_fifo` appears:
 - Right-click → **Remove File from Project**

If it does NOT appear → that's fine.

3 Recreate the IP completely fresh

In Vivado:

1. Open **IP Catalog**
2. Search: **FIFO Generator**
3. Double-click
4. Configure your FIFO
5. **Set name:**

```
nginx
```

[Copy code](#)

```
dma_fifo
```

6. Click **OK**
7. Vivado will ask to generate files → Click **Generate Output Products**

Vivado will now create a clean directory:

```
pgsql
```

[Copy code](#)

```
dma_fifo.xci  
component.xml  
dma_fifo.vhd
```

4 Verify with Tcl

Run:

```
tcl
```

 Copy code

```
get_ips
```

You should now see:

```
nginx
```

 Copy code

```
dma_fifo  
mmap_data_fifo  
mmap_request_fifo
```

Now the JSON schema error disappears.

! Why deleting the folder is necessary

The error explicitly says the JSON file is missing mandatory fields:

```
nginx
```

 Copy code

```
gen_directory
```

This means your `.xci` (or the JSON inside it) is **corrupted**, likely because:

- Vivado tried to generate inside an existing folder and failed
- Old and new metadata got mixed
- There was a partial IP generation previously
- You copied a folder from someone else without regenerating

Vivado **cannot repair** corrupted metadata; the only fix is regeneration.

★ If you want, send me:

```
bash
```

[Copy code](#)

```
ls ../../dram_test_1.0/src/
```

and I will tell you exactly which folders are valid IPs and which are corrupted.

[Copy](#) [Like](#) [Comment](#) [Share](#) [Refresh](#) [More](#)

Please, how to go about this error?

No problem — let's fix it.

Before I give the correct steps, I need to see **what Vivado is actually seeing** and **which file is corrupted**.

✅ I need two things from you so I can diagnose the error correctly:

👉 1. Tell me the exact path where the error happened

What is the folder that contains the corrupted IP?

Run this in your terminal and paste the output:

```
bash
```

[Copy code](#)

```
ls /home/UFAD/audreycyriell.mo/Documents/reconfig1/final_project/provided_
```

This will show all IP folders (including `dma_fifo` or others) so I can see which one broke.

👉 2. Tell me what Vivado thinks the IPs are

In the Vivado Tcl Console, run:

tcl

 Copy code

```
get_ips
```

You already ran this before and got:

nginx

 Copy code

```
mmap_data_fifo mmap_request_fifo
```

Please run it again **after** the JSON error to make sure it's the same.

WHY I NEED THIS

The JSON schema error:

vbnet

 Copy code

```
missing: [ "gen_directory" ]
```

means:

- Vivado saw a **broken IP**
- The **.xci file inside that IP's folder is corrupted**
- Vivado cannot regenerate it because the JSON is invalid
- Vivado cannot load it because the metadata is incomplete
- Vivado cannot overwrite it because the folder already exists

To fix it, we need to:

- ✓ Identify **which IP folder is corrupted**
- ✓ Delete only that IP folder (not the whole project)
- ✓ Recreate the IP correctly using the IP Catalog

But I cannot tell you exactly which folder to delete until I see the listing.

 **As soon as you send the output of:**

bash

 Copy code

```
ls /home/UFAD/.../dram_test_1.0/src/
```

I will tell you:

- Which folder is corrupted
- Exactly what to delete
- How to regenerate it
- How to remove the JSON error permanently

💡 **100% solvable — we just need to locate the bad IP folder.**

📄 👍 💬 ↗️ ↺️ ...

how to uninstall vivado?

Here are **all the correct ways to completely uninstall Vivado** depending on how you installed it (Installer GUI or Unified Installer).

I'll give you the **simple method first**, then the **full clean removal** if needed.

✅ **METHOD 1 — Use Xilinx's built-in uninstaller (recommended)**

Open a terminal and run:

```
bash
```

📄 Copy code

```
sudo /opt/Xilinx/Vivado/*/uninstall
```

Example (for Vivado 2022.2):

```
bash
```

📄 Copy code

```
sudo /opt/Xilinx/Vivado/2022.2/uninstall
```

It will launch the Xilinx uninstaller and remove:

- Vivado
- DocNav
- Cables drivers (if selected)
- Common libraries installed with that version

METHOD 2 — If you used Xilinx Unified Installer

Run:

```
bash
```

 Copy code

```
sudo /opt/Xilinx/uninstall
```

This removes:

- Vivado
- Vitis
- Cable drivers
- Petalinux (if installed through unified installer)
- All shared runtimes under /opt/Xilinx

METHOD 3 — If the uninstaller is missing (corrupted installation)

Step 1 — Delete Xilinx folders manually:

```
bash
```

 Copy code

```
sudo rm -rf /opt/Xilinx  
sudo rm -rf ~/Xilinx  
sudo rm -rf ~/.Xilinx
```

Step 2 — Remove desktop shortcuts:

```
bash
```

 Copy code


```
rm -f ~/Desktop/Vivado*.desktop  
rm -f ~/Desktop/DocNav*.desktop  
rm -f ~/Desktop/Vitis*.desktop
```

Step 3 — Remove udev rules for Xilinx cable drivers:

bash

 Copy code

```
sudo rm -f /etc/udev/rules.d/52-xilinx-digilent-usb.rules  
sudo rm -f /etc/udev/rules.d/52-xilinx-pcusb.rules  
sudo rm -f /etc/udev/rules.d/99-xilinx-digilent-usb.rules  
sudo rm -f /etc/udev/rules.d/99-xilinx-pcusb.rules  
sudo udevadm control --reload-rules
```

METHOD 4 — Full clean wipe (if Vivado is very broken)

Use *only* if Vivado refuses to uninstall or is corrupted.

bash

 Copy code

```
sudo rm -rf /opt/Xilinx  
sudo rm -rf /usr/local/share/Xilinx  
sudo rm -rf ~/.Xilinx  
sudo rm -rf ~/Xilinx  
sudo rm -rf ~/.cache/Xilinx  
sudo rm -rf ~/.config/Xilinx
```

This removes **every trace** of Vivado.

Optional — Remove environment variables from .bashrc

Edit your bashrc:

bash

 Copy code

```
gedit ~/.bashrc
```

Remove lines like:

```
bash
```

 Copy code

```
source /opt/Xilinx/Vivado/2022.2/settings64.sh
export PATH=/opt/Xilinx/Vivado/2022.2/bin:$PATH
```

Save + close, then reload:

```
bash
```

 Copy code

```
source ~/.bashrc
```

Want me to help reinstall Vivado cleanly?

I can give you:

- correct installer link
- installation steps
- how to install without errors
- how to set up cable drivers properly
- how to set up license correctly

Just tell me the version you need (e.g., 2020.2, 2021.1, 2022.2).

     ...

```
/Xilinx$ ls
DocNav Model_Composer Vitis Vitis_HLS Vivado xic
```

Perfect — that folder listing confirms you installed Vivado (and other Xilinx tools) in the default location:

```
bash
```

 Copy code

/opt/Xilinx/

So now you can uninstall cleanly.

Below is the **correct uninstall procedure for your setup**.

✓ 1. Uninstall Vivado using the built-in uninstaller

Run this in your terminal:

```
bash
```

 Copy code

```
sudo /opt/Xilinx/Vivado/*/uninstall
```

Example (if your version is 2023.1):

```
bash
```

 Copy code

```
sudo /opt/Xilinx/Vivado/2023.1/uninstall
```

This will start the official Xilinx uninstaller.

Repeat for every Vivado version inside /opt/Xilinx/Vivado/ .

✓ 2. Uninstall Vitis (if installed)

```
bash
```

 Copy code

```
sudo /opt/Xilinx/Vitis/*/uninstall
```

✓ 3. Uninstall Model Composer / HLS / DocNav (if needed)

```
bash
```

 Copy code

```
sudo /opt/Xilinx/Model_Composer/*/uninstall
sudo /opt/Xilinx/Vitis_HLS/*/uninstall
sudo /opt/Xilinx/DocNav/*/uninstall
```

! If any uninstaller fails or folder looks corrupted

then you must delete the folders manually:

```
bash
```

[Copy code](#)

```
sudo rm -rf /opt/Xilinx
sudo rm -rf ~/Xilinx
sudo rm -rf ~/.Xilinx
```

This removes **all** Xilinx software.

4. Clean up udev cable drivers (optional but recommended)

```
bash
```

[Copy code](#)

```
sudo rm -f /etc/udev/rules.d/52-xilinx-digilent-usb.rules
sudo rm -f /etc/udev/rules.d/52-xilinx-pcusb.rules
sudo udevadm control --reload-rules
```

5. Remove environment variables from your shell

Open `.bashrc` :

```
bash
```

[Copy code](#)

```
gedit ~/.bashrc
```

Delete any lines like:

```
bash
```

```
source /opt/Xilinx/Vivado/2023.1/settings64.sh
export PATH=/opt/Xilinx/Vivado/2023.1/bin:$PATH
```

Save and reload:

```
bash
```

 Copy code

```
source ~/.bashrc
```

After this, your system is completely clean

No Vivado, no Vitis, no hidden configs, no leftover IP folders.

? Do you want help reinstalling Vivado cleanly?

I can tell you:

- which version is safest for your FPGA board
- how to install cable drivers correctly
- how to install without admin permission errors
- how to set up LM_LICENSE_FILE

Just tell me your **Vivado version** or your **FPGA board model**.

     ...

```
/Xilinx/Vivado/2024.2$ ls
bin  doc   gnu   lib           product_rel_win.log
settings64.csh tps      vivado_30345.backup.jou
vivado_9888.backup.log xsct-trim
cli  examples ids_lite platforms  reportstrategies
settings64.sh  vivado_24150.backup.jou
vivado_30345.backup.log vivado.jou
data fonts  include  product_rel_lin.log scripts
```

strategies vivado_24150.backup.log
vivado_9888.backup.jou vivado.log

To **fully uninstall Vivado 2024.2** (and any other Xilinx tools) from Linux, you need to:

✓ Method 1 — Use the Official Uninstaller (Recommended)

Xilinx installs an uninstaller inside each tool folder.

Run this command:

```
bash
```

[Copy code](#)

```
sudo /Xilinx/Vivado/2024.2/bin/uninstall
```

If `uninstall` is not executable, enable permissions:

```
bash
```

[Copy code](#)

```
sudo chmod +x /Xilinx/Vivado/2024.2/bin/uninstall  
sudo /Xilinx/Vivado/2024.2/bin/uninstall
```

It will open a text GUI asking which components to remove.

✓ Method 2 — Use the Universal Xilinx Uninstaller

There is also a global uninstaller:

```
bash
```

[Copy code](#)

```
sudo /Xilinx/uninstall
```

or inside the main folder:

```
bash
```

[Copy code](#)

```
sudo /Xilinx/uninstall.sh
```

✗ If no uninstaller exists: Manual Removal

1 Delete the installation directory

Be careful and run:

```
bash
```

[Copy code](#)

```
sudo rm -rf /Xilinx/Vivado/2024.2
```

If you want to remove everything Xilinx installed:

```
bash
```

[Copy code](#)

```
sudo rm -rf /Xilinx
```

2 Remove environment variables added to your shell

Check your ~/.bashrc or ~/.zshrc :

```
bash
```

[Copy code](#)

```
nano ~/.bashrc
```

Remove lines like:

```
bash
```

[Copy code](#)

```
source /Xilinx/Vivado/2024.2/settings64.sh
```

```
source /Xilinx/Vitis/2024.2/settings64.sh
```

Save and reload:

```
bash
```

[Copy code](#)

```
source ~/.bashrc
```

3 Remove desktop entries (optional)

bash

 Copy code

```
rm ~/.local/share/applications/vivado.desktop  
rm ~/.local/share/applications/vitis.desktop
```

? If you want, send me the result of:

bash

 Copy code

```
ls /Xilinx/Vivado/2024.2/bin | grep uninstall
```

I'll tell you exactly which uninstaller file is present.

     ...

```
ls  
bootgen          cs_server hw_server  loader  
program_ftdi     setupEnv.sh symbol_server  
unwrapped vlm  xelab   xsc  xsim.tcl xvhdl  
cdoutil          diffbd  isoutil  manage_ipcache  
rdiArgs.sh       stapl_player tcflog    updatemem xar  
xlicdiag  xsdb  xtclsh  xvlog  
combine_dfx_bitstreams dtc    ldlibpath.sh  
pdi_dbg_util  setEnvAndRunCmd.sh svf_utility  
unsetldlibpath.sh vivado  xcrp  xrt_server xsim  
xvc_pcie
```

It looks like **there is no built-in uninstaller** inside:

```
/Xilinx/Vivado/2024.2/bin/
```

Some Vivado versions include an `uninstall` script, but **2023+ versions often don't**, depending on the install method.

So you will uninstall manually — but safely.
Here is the **correct full procedure** for Linux.

✓ Step 1 — Remove the Vivado directory

Delete only the Vivado 2024.2 folder:

```
bash
```

[Copy code](#)

```
sudo rm -rf /Xilinx/Vivado/2024.2
```

If you also want to remove **Vitis**, **DocNav**, etc., tell me and I will give the safe commands.

✓ Step 2 — Remove environment variables

Open your shell configuration:

```
bash
```

[Copy code](#)

```
nano ~/.bashrc
```

Look for lines like:

```
bash
```

[Copy code](#)

```
source /Xilinx/Vivado/2024.2/settings64.sh
```

```
source /Xilinx/Vitis/2024.2/settings64.sh
```

Delete them.

Reload your shell:

```
bash
```

[Copy code](#)

```
source ~/.bashrc
```

✓ Step 3 — Remove desktop shortcuts (optional)

Run:

bash

 Copy code

```
rm -f ~/.local/share/applications/vivado*.desktop  
rm -f ~/.local/share/applications/vitis*.desktop
```

If you want to fully remove all Xilinx tools

Run:

bash

 Copy code

```
sudo rm -rf /Xilinx
```

This deletes Vivado, Vitis, Model Composer, HLS, DocNav, everything.

If you want, send me the output of:

bash

 Copy code

```
ls /Xilinx
```

I will give you the exact removal commands for each tool.

     ...

```
ls Xilinx/  
DocNav Model_Composer Vitis Vitis_HLS Vivado xic
```

Here are the **safe uninstall commands** for every tool inside your `/Xilinx` directory.

1. Uninstall Vivado

```
bash
```

 Copy code

```
sudo rm -rf /Xilinx/Vivado
```

2. Uninstall Vitis

```
bash
```

 Copy code

```
sudo rm -rf /Xilinx/Vitis
```

3. Uninstall Vitis HLS

```
bash
```

 Copy code

```
sudo rm -rf /Xilinx/Vitis_HLS
```

4. Uninstall Model Composer

```
bash
```

 Copy code

```
sudo rm -rf /Xilinx/Model_Composer
```

5. Uninstall DocNav

```
bash
```

 Copy code

```
sudo rm -rf /Xilinx/DocNav
```

6. Uninstall xic (Xilinx Installer Cache)

This is optional, but recommended if you want a *clean* new install:

```
bash
```

 Copy code

```
sudo rm -rf /Xilinx/xic
```

7. Remove environment variables

Open:

```
bash
```

 Copy code

```
nano ~/.bashrc
```

Delete any lines like:

```
bash
```

 Copy code

```
source /Xilinx/Vivado/.../settings64.sh
```

```
source /Xilinx/Vitis/.../settings64.sh
```

Save, then reload:

```
bash
```

 Copy code

```
source ~/.bashrc
```

8. Remove desktop shortcuts (optional)

```
bash
```

 Copy code

```
rm -f ~/.local/share/applications/vivado*.desktop
```

```
rm -f ~/.local/share/applications/vitis*.desktop
```

After this

Your system is completely cleaned and ready for a fresh reinstall.

If you want, I can also guide you on:

- ✓ Installing **Vivado 2024.2** cleanly
- ✓ Installing **Vivado 2023.1** (most stable)
- ✓ Setting your PATH properly
- ✓ Fixing any install errors during reinstall

Just tell me!



```
export GTK_PATH=/usr/lib/gtk-3.0
export SYSTEMC_HOME=/usr/local/systemc
export
LD_LIBRARY_PATH=$SYSTEMC_HOME/lib:$LD_LIBRARY_
PATH
export
CPLUS_INCLUDE_PATH=$SYSTEMC_HOME/include:$CPL
US_INCLUDE_PATH
source
/home/audrey/Xilinx/Vivado/2024.2/settings64.sh
```

Here's what each of these lines does and what you need to adjust after uninstalling and reinstalling Vivado.

✓ Meaning of each line

1. GTK_PATH

bash

Copy code

```
export GTK_PATH=/usr/lib/gtk-3.0
```

You probably needed this for GUI issues.

✓ Safe to keep — it doesn't depend on Vivado.

2. SYSTEMC_HOME

You added SystemC manually:

bash

 Copy code

```
export SYSTEMC_HOME=/usr/local/systemc
export LD_LIBRARY_PATH=$SYSTEMC_HOME/lib:$LD_LIBRARY_PATH
export CPLUS_INCLUDE_PATH=$SYSTEMC_HOME/include:$CPLUS_INCLUDE_PATH
```

✓ These are correct

✓ Keep them because they are unrelated to Vivado

3. Vivado environment setup

bash

 Copy code

```
source /home/audrey/Xilinx/Vivado/2024.2/settings64.sh
```

This loads Vivado binaries, libraries, JTAG drivers, etc.

⚠ **If you deleted Vivado, this path no longer exists.**

If you leave it, every new terminal will show an error:

yaml

 Copy code

```
No such file or directory: settings64.sh
```

! What you must do now

Open `.bashrc` :

bash

 Copy code

```
nano ~/.bashrc
```

Find this line:

```
bash
```

 Copy code

```
source /home/audrey/Xilinx/Vivado/2024.2/settings64.sh
```

Delete it.

Save and reload:

```
bash
```

 Copy code

```
source ~/.bashrc
```

After reinstalling Vivado

When you reinstall Vivado (2024.2 or earlier), update the new path:

Example if the new install is:

```
swift
```

 Copy code

```
/tools/Xilinx/Vivado/2024.2/
```

Then add:

```
bash
```

 Copy code

```
source /tools/Xilinx/Vivado/2024.2/settings64.sh
```

or whatever path the installer gives you.

If you want help:

- ✓ Installing Vivado properly
- ✓ Choosing the most stable version
- ✓ Fixing errors during installation
- ✓ Setting PATH variables correctly

Just tell me!



```
sudo: ./Xilinx_Unified_2021.2_1021_0703_Lin64.bin:  
command not found
```

This error means **the file is not executable OR you are not in the correct directory.**

Let's fix it step by step.

✓ 1. Make sure you are inside the correct folder

Run:

```
bash
```

Copy code

```
ls
```

You should see:

```
python
```

Copy code

```
Xilinx_Unified_2021.2_1021_0703_Lin64.bin
```

If not, navigate to the folder where it is:

Example:

```
bash
```

Copy code


```
cd ~/Downloads
```

or:

```
bash
```

[Copy code](#)

```
cd /home/audrey/Downloads
```

Then:

```
bash
```

[Copy code](#)

```
ls
```

✓ 2. Make the installer executable

Once you see the file in `ls`, run:

```
bash
```

[Copy code](#)

```
chmod +x Xilinx_Unified_2021.2_1021_0703_Lin64.bin
```

✓ 3. Run the installer correctly

Use:

```
bash
```

[Copy code](#)

```
sudo ./Xilinx_Unified_2021.2_1021_0703_Lin64.bin
```

If it still says **command not found**, try specifying the current directory explicitly:

```
bash
```

[Copy code](#)

```
sudo bash ./Xilinx_Unified_2021.2_1021_0703_Lin64.bin
```

or:

```
bash
```

```
sudo sh ./Xilinx_Unified_2021.2_1021_0703_Lin64.bin
```

Most common reasons for “command not found”

Wrong folder

You're not inside the directory containing the installer.

Wrong filename

Make sure the name is **exact**.

Corrupted download

If the file is very small (e.g., 0 KB), it's broken. Check:

```
bash
```

 Copy code

```
ls -lh
```

The installer should be around **40–60 GB** depending on options.

If you want, send me the output of:

```
bash
```

 Copy code

```
pwd
```

```
ls -lh
```

and I'll tell you exactly what to do.

Thank you Professor for your recommendation. I have uninstalled Vivado 2023.1 to installed 2021 .
Unfortunately this version is not support my devices

that run on Ubuntu 22.04. It's been many years that I haven't used Windows on my computer. I am wondering whether I should use a VM on my system in order to install the Ubuntu version that supports Vivado 2023.1

Here's the **clear and honest guidance** based on your situation, your Ubuntu version, and Vivado compatibility.

✓ TL;DR — Do NOT use a VM for Vivado. Install a compatible Vivado version instead.

Let me explain in detail so you can make the best decision.

✗ 1. Do NOT run Vivado in a Virtual Machine (VM)

Vivado is very heavy and uses:

- GPU acceleration
- Kernel-level JTAG USB drivers
- High-performance Tcl GUIs
- Large memory and storage footprint

In a VM, these features break or run extremely slow.

Also, USB-JTAG does *not* work properly in VirtualBox or VMware.

🚫 **Vivado + VM = bad experience + many errors**

! 2. Vivado 2021.x does NOT support Ubuntu 22.04

Correct — Vivado 2021.x supports:

- Ubuntu 18.04
- Ubuntu 20.04

Not 22.04.

That's why your installer failed.

★ **3. Your BEST solution: Install Vivado 2023.1 (or 2024.2) which supports Ubuntu 22.04**

✓ **Vivado 2023.1**

Officially supports Ubuntu 22.04

This is the recommended version for your system.

✓ **Vivado 2024.2**

Also supports Ubuntu 22.04

Modern, stable, and recommended by AMD/Xilinx for new work.

👍 **4. You should NOT downgrade your OS or use VM**

You currently use:

- ♦ *Ubuntu 22.04 (LTS)*

This is perfect.

Keep your OS as it is.

Instead, install a Vivado version that matches your OS:

Ubuntu version	Supported Vivado versions
22.04	Vivado 2023.1, 2024.1, 2024.2
20.04	2020.3 — 2022.2
18.04	2018.3 — 2020.2

🔥 **Recommended Action for You**

● **Install Vivado 2023.1 (best match for 22.04)**

Or

Install Vivado 2024.2 (newest + stable)

You will have:

- Full driver compatibility
- No GUI crashes
- No USB-JTAG issues
- Faster synthesis
- Better IP Catalog stability

As your professor-style helper, my recommendation:

Reinstall Vivado 2023.1

This is the version your project already uses, and it works perfectly with Ubuntu 22.04.

If you want, I can help you with:

- ✓ Clean uninstall (removing leftover paths)
- ✓ Download links for Vivado 2023.1
- ✓ Correct installation steps for Ubuntu 22.04
- ✓ Fixing drivers (cable drivers, udev rules)
- ✓ Verify environment variables (`settings64.sh`)

Just tell me!



reformulate this for me please:

Thank you Professor for your recommendation. I have uninstalled Vivado 2023.1 to installed 2021 .

Unfortunately this version is not support my devices that run on Ubuntu 22.04. It's been many years that I haven't used Windows on my computer. I am wondering

whether I should used a VM on my system in other to install the ubuntu version that supports Vivado 2023.1

Here is a clearer and more polished version:

Thank you, Professor, for your recommendation. I uninstalled Vivado 2023.1 to install the 2021 version, but unfortunately that version does not support my device, as I am running Ubuntu 22.04. I have not used Windows on my computer for many years, so I am wondering whether I should use a virtual machine in order to install an Ubuntu version that is compatible with Vivado 2023.1.

If you'd like it more formal, more concise, or more friendly, I can adjust it!



quelle est le resultat de la borne superieure de la division de 5 par 3

La *borne supérieure* de la division de 5 par 3 correspond au **plafond** (le *ceiling*), c'est-à-dire la plus petite valeur entière $\geq 5/3$.

$$\frac{5}{3} \approx 1,666 \dots$$

La borne supérieure est donc :

$$\lceil 5/3 \rceil = 2$$

✅ **Résultat : 2**



meme question pour 4 par 3

On cherche la **borne supérieure** (plafond) de 4 divisé par 3 :

$$\frac{4}{3} \approx 1,333 \dots$$

La plus petite valeur entière $\geq 1,333\dots$ est :

$$\lceil 4/3 \rceil = 2$$

✓ **Résultat : 2**



et 6/3

Pour $\frac{6}{3}$:

$$\frac{6}{3} = 2$$

La **borne supérieure** (plafond) est la plus petite valeur entière ≥ 2 , donc :

$$\lceil 6/3 \rceil = 2$$

✓ **Résultat : 2**



To support any window size, the controller determines how far to slide the maximum-sized window across the image. For example, if the FPGA provides a 10×10 window, but the user requests a 3×3 window, the controller would slide the 10×10 window seven pixels

past the right edge and bottom edge of the image. The controller pads all unused window elements with 0, which is done automatically on reset. ?

what does this mean? I don't understand this example. Can you please explain it into details?

Sure! Let's go **step by step** — this is about **sliding windows in FPGA image processing**. I'll break it down carefully.

1 Context: Sliding window on an image

- Suppose you have an image, e.g., 10×10 pixels.
- A **window** is a smaller sub-region that moves (slides) across the image.
- For example, a 3×3 window could slide over the image to apply an operation (like convolution or edge detection).

2 The FPGA always provides a maximum-sized window

- In your example, the FPGA hardware always gives a **10×10 window** (the “maximum-sized” window it supports).
- But the user might only want a **smaller window**, e.g., 3×3.

So the hardware window is bigger than the user window.

3 How the controller handles this

The controller must **adapt the hardware maximum window to any requested window size**:

- User requested: 3×3
- Hardware window: 10×10

The hardware always outputs 10×10 pixels for each position.

The **controller slides this large window across the image**, but **only uses the top-left**

3×3 portion.

4 Example: sliding the 10×10 window for a 3×3 request

- Image: let's say 10×10 pixels.
- User wants a 3×3 window.

The hardware window is 10×10, so when you want to move the 3×3 “virtual” window across the image:

- Horizontally, you move the window from left to right.
- Vertically, you move it from top to bottom.
- Since the hardware window is bigger than the requested window, **most of the hardware window extends past the image edges.**

How far does it extend?

$$\text{extra pixels} = \text{hardware window size} - \text{user window size} = 10 - 3 = 7$$

- So the controller slides the 10×10 window **7 pixels past the right edge** and **7 pixels past the bottom edge.**
- These “extra” pixels are **not part of the image** → the controller fills them with 0 (padding).

5 Padding

- The “unused elements” of the 10×10 window (outside the 3×3 region) are automatically filled with 0.
- This ensures that the convolution or other operation behaves correctly at edges.

✓ Summary in simpler terms

1. FPGA always gives the biggest window it can (10×10).
2. User can request any smaller window (3×3 in this example).
3. The controller slides the 10×10 window so that the smaller user window covers the image completely.
4. Extra parts of the window (outside the image) are **padded with 0** automatically.

If you want, I can **draw a little diagram** showing how the 10×10 hardware window slides over a 3×3 requested window on a small image — it makes it much easier to visualize.

Do you want me to do that?



I want to build a new sliding window architecture using systolic array and in the state of the art, I found a paper can you please go over the paper and explain to me what it does?

Of course — I can explain the paper in detail, but I need **either the title, the PDF, a link, or the text of the paper** so I can read it.

Please upload the paper or paste the title here, and I'll give you:

- ✓ A clear summary
- ✓ The main contributions
- ✓ How the architecture works
- ✓ How it relates to your systolic sliding-window design
- ✓ Strengths, limitations, and what you can reuse

Just send me the paper.



fpga081-stitt.pdf
PDF


This is the paper

Thought for 15s >

Great — I read the paper you uploaded and went through the important sections for you. Below I give a clear, structured explanation of **what the paper does**, how the proposed **window generator** works (the key building blocks and tricks), the **performance / trade-offs**, and **concrete takeaways** for building a sliding-window systolic array.


I cite the paper where I draw technical details.  fpga081-stitt

1) What problem the paper solves (short)

The paper targets **scalable sliding-window generation** on FPGAs so you can produce many windows in parallel (p windows per cycle) without huge timing or padding overheads. The motivation: modern FPGAs (and server FPGA systems) have much larger memory bandwidth, but existing sliding-window generators don't scale to exploit that bandwidth — they either require padding, lots of muxes, or become timing-limited when you replicate many pipelines. The paper presents a window generator that scales to >330 GB/s ($\approx 8.7\times$ better than prior work) and supports runtime-configurable window and image sizes with no input padding overhead.  fpga081-stitt

2) High-level architecture (three blocks)

The window generator is composed of three main blocks:

1. **Variable-Read FIFO (VRF)** — handles variable read sizes from the incoming pixel stream so you don't need to pad rows to multiples of p . It lets the system read a variable number of pixels each cycle from the FIFO (e.g., read 3, then 1, then 3, then 1 when $p=3$) to precisely match row endings without software padding. This saves expensive preprocessing and PCIe transfer time.
2. **Window Buffer (chained row FIFOs)** — a chain of w_r FIFOs (one FIFO per row of the maximum window height). The buffer writes incoming p -pixel words into the bottom FIFO and then, once a row is full, shifts data upward (read and write into the next FIFO). This produces aligned columns of pixels for the coalescer without expensive muxing.  fpga081-stitt
3. **Window Coalescer** — receives p columns each cycle from the window buffer and assembles them into approximately p complete windows per cycle by shifting




register columns. It avoids large mux networks and register-enable fanout by controlling when it starts reading (so all inputs are valid during a row) and by using purely local shifts/registers inside the coalescer.  fpga081-stitt

Diagram references and more detail are in Figures 1–3 of the paper.  fpga081-stitt

3) Two key hardware techniques that enable scalability

These are the useful engineering ideas you should note:


- **Barrel-shifter alignment in the VRF**

Instead of using $p \times p:1$ muxes (which blow up the critical path), the VRF uses a *pipelined barrel shifter* to realign outputs based on an output index. That makes the critical path essentially independent of p (at the cost of a few cycles latency), enabling much higher clock rates as p grows.  fpga081-stitt


- **No global enable / no giant mux fanout in the coalescer**

Earlier designs used enable signals (fanout to many registers) or big mux trees for alignment. The paper's controller waits until all necessary FIFO data exists and then performs reads so that the coalescer can run without per-register enables.

Removing that enable fanout is a major timing payoff — propagation delays become independent of p .  fpga081-stitt

Other helpful measures: using chained row FIFOs that shift data upward (so the top FIFO always corresponds to the top row) and relying on register duplication (Quartus replicates high-fanout registers) to control fanout and timing.  fpga081-stitt

4) How the generator handles image edges and different window sizes

- The design supports **runtime-configurable** window ($w_r \times w_c$) and image sizes.
- To handle smaller windows while supporting a fixed maximum hardware window, the controller **slides the maximum window across the image** and **zero-pads unused window elements** on reset. (Example from paper: if hardware supports 10×10 but user asks for 3×3 , the controller slides the 10×10 window 7 pixels past right/bottom edges and uses 0s for the unused elements.) This keeps generation time similar for small windows (except extra startup to fill FIFOs).  fpga081-stitt

- To avoid software padding, the VRF and window buffer mark invalid pixels when less than p remain in a row and treat them accordingly so windows remain correct.

📄 fpga081-stitt

5) Performance and resource tradeoffs (what they show)

- **Scalability:** they demonstrate pipeline replication up to 1024 pipelines and measured maximum clock frequencies much higher than previous work for $p \geq 64$. Max memory bandwidth reached ≈ 336 GB/s vs ≈ 38.4 GB/s for the older approach ($\approx 8.7\times$ improvement). 📄 fpga081-stitt
- **Convolution benchmarks:** on Intel Broadwell + Arria 10 (shared memory system), FPGA solution outperforms a 12-core Xeon and a high-end Nvidia P6000 for **traditional convolution** (one filter per image) and is much more energy efficient. For CNN scenarios, they project that with enough bandwidth (e.g., Stratix 10), FPGA could beat GPUs for many common parameter sets.
- **Resources:** compared to prior work the new design uses fewer LUTs (avg $\approx 40\%$ less), but FFs increase ($\sim 20\%$ more) and BRAM usage also increases ($\sim 40\%$ more) — a conscious tradeoff (use more storage/registers to remove muxes and meet timing). 📄 fpga081-stitt

6) Why this matters for your systolic-array sliding-window design

If your compute stage is a **systolic array** (a PE array that expects windows or columns fed in a specific rhythm), this window generator is a very good match:







- **Feeds many parallel windows per cycle (p)** — exactly what you want to sustain a wide systolic array feeding many PEs in parallel. The coalescer outputs p windows per cycle already shaped as register columns and can be mapped to the PE inputs.
- **Avoids padding and pre-processing** — useful when your systolic array is sized for some p and you want to avoid software padding overheads.
- **Timing-friendly for large p** — barrel-shifter alignment and elimination of global enables means the window generator can run at high frequency even when p is

large, which helps keep a high-frequency systolic array fed.




- **Border handling** — the controller's strategy of sliding the maximum window and zero-padding unused elements gives a predictable, hardware-only border handling method that your systolic array can accept without extra logic.

7) Practical implementation notes & suggestions for you




(Concrete tips if you plan to adapt/implement parts of this paper)

1. **Adopt the VRF idea** — your input stage should allow variable reads from the FIFO to avoid row padding. Use a pipelined barrel shifter for output alignment rather than $p \times p$ muxes.  [fpga081-stitt](#)
2. **Row FIFOs that shift upward** — implement w_r row FIFOs and the “write to bottom; read and push upward” mechanism so the top FIFO always holds the top window row. That removes index muxing headaches.  [fpga081-stitt](#)
3. **Design controller carefully** — the controller sequences when to start outputting columns so coalescer inputs remain valid for the entire row; this removes enable fanout and improves timing.  [fpga081-stitt](#)
4. **Match coalescer output format to your systolic array input** — the coalescer naturally produces window columns; map each register column to a column (or group of inputs) of your PE array to avoid reformatting cost.
5. **Be ready to trade area for timing** — expect to use more FFs and BRAM to eliminate muxes and meet timing. Plan for register duplication where high fanout exists.  [fpga081-stitt](#)
6. **Memory bandwidth is the primary bottleneck** — replication of your compute (systolic array) will be limited by memory bandwidth. Use the window generator to scale up to what bandwidth you have, but plan profiling and projections (paper shows saturation points where more replication stops helping).  [fpga081-stitt](#)
7. **Check the open-source repo** — authors released a `window_gen` implementation at https://github.com/ARC-Lab-UF/window_gen (paper references it). That repo is an excellent starting point to study RTL and adapt pieces.  [fpga081-stitt](#)

8) Limitations & what to watch out for

- **Increased FF/BRAM usage** — if your FPGA is resource constrained, you might need to balance p and window sizes to fit.  [fpga081-stitt](#)
- **Startup latency** — filling the extra row FIFOs adds initial latency before you can stream full rows. For streaming continuous inputs this is amortized; for lots of tiny images it can matter.  [fpga081-stitt](#)
- **Register-duplication behavior is vendor/tool dependent** — the paper uses Quartus/Arria10 and relies on register replication heuristics; on other FPGAs (Xilinx) you may need manual duplication or floorplanning to get the same timing benefits.  [fpga081-stitt](#)

9) Suggested next steps (actionable)

1. Clone the authors' `window_gen` repo and examine VRF, window buffer, and coalescer RTL.  [fpga081-stitt](#)
2. Prototype a **small p** (e.g., $p=4$ or 8) hardware model that feeds your systolic array PEs to validate interface and timing.
3. Replace any mux/alignment in your current design with a barrel-shifter based alignment block from the VRF idea.  [fpga081-stitt](#)
4. Synthesize on your target FPGA and measure how many replications you can support before memory bandwidth saturates — this will guide p and replication choices.  [fpga081-stitt](#)
5. If you use Xilinx rather than Intel/Quartus, plan extra experiments for register replication/fanout, since the vendor tools differ; you might need to explicitly duplicate registers to reduce fanout.

10) Short one-paragraph summary you can use in a report

The paper presents a scalable hardware window generator (VRF + window buffer + coalescer) that generates p sliding windows per cycle without padding or large mux fanouts. It uses a pipelined barrel-shifter for alignment and a FIFO chaining strategy to avoid alignment muxes, enabling replication up to >330 GB/s and high clock rates

for large p. Evaluations on Broadwell+Arria10 show significant speedups vs CPU/GPU for traditional convolution and favorable projections for CNNs given sufficient bandwidth.

If you want, I can **do one of the following next** (pick one and I'll proceed):

- Walk through the **VRF RTL** in the repo and explain each module line-by-line (if you want to adopt it).
- Sketch how to **connect the coalescer outputs to your systolic PE array** (show data layout and timing).
- Prototype a **small Verilog/VHDL template** (p=4, 3×3 window) that feeds a tiny systolic array so you can test the flow on your board.

Which of those would help you most right now?

Sources



Does this promote data reuse from the systolic array aspect?

Great question — *this is exactly where people get confused*, because the paper focuses on **window generation**, not on the compute array itself.

Let's make the distinction very clear:

✓ Short answer:

YES, the architecture *indirectly* promotes data reuse for a systolic array — **but NO**, the window generator itself does *not* perform reuse internally.

Instead, it **feeds the systolic array in the perfect reuse-friendly format** so the systolic array *can* exploit reuse efficiently.

Let me explain in detail.

1 What kind of data reuse does a systolic array need?

A 2-D convolution systolic array usually relies on:

Pixel reuse

A single pixel participates in multiple overlapping windows.

Example:

For a 3×3 filter, each pixel is reused **9 times**.

Weight reuse

Each filter weight is reused across every window in a row/column.

Spatial reuse along rows/columns

Neighboring PEs share partial sums or activations.

For a systolic array to benefit, the input data must arrive:

- **in a sliding-window pattern**
- **with minimal duplication**
- **with correct timing offsets**
- **in column-major or row-major streams that match PE interconnect**

If the window generator produces windows with too much duplication,

OR

if windows must be rearranged in compute stage → you lose reuse.

2 What this paper's window generator does for reuse

The window generator outputs **p windows per cycle**, but **each window is produced with minimal duplication**, because:

✓ It creates windows by shifting rows stored in row-FIFOs

→ this avoids reloading rows from memory

→ **pixel reuse occurs inside the buffering structure**

✓ It streams the window column by column, neatly aligned

This is EXACTLY what a systolic array expects if each column goes into a PE column.

✓ **It eliminates padding/muxing overhead**

- removes unnecessary data movement
- preserves reusable data in place
- reduces bandwidth waste

✓ **It keeps window elements stationary once in the coalescer**

- no extra copying, no extra index computations

****So the structure does NOT explicitly “reuse data like a systolic array.”**

But it delivers data in a format that ALLOWS the systolic array to reuse efficiently.**

3 What the paper does *not* do

The window generator **does not implement:**

- PE-to-PE systolic propagation
- partial-sum accumulation
- compute flow scheduling
- local broadcast of weights
- temporal reuse loops

It is not a systolic architecture.

It is a **data preparation / data streaming engine**.

4 So does it promote reuse?

✓ **YES — it promotes pixel reuse operationally:**

The architecture stores only w_r rows and shifts them upward; pixels are read **once** and used in many windows.

This is **exactly how systolic arrays expect data to be reused**.

✓ **YES — it promotes spatial reuse:**

The coalescer produces columns aligned to systolic PE columns.

✓ YES — it minimizes redundant reads, extra padding, and memory accesses

✗ NO — it does *not* perform systolic style propagation

That happens in your PE array.

5 How you connect this to a systolic array

To turn this into a systolic-convolution engine:

- Each column output from the coalescer → goes to a **column of PEs**
- Each PE receives:
 - One pixel value from above
 - One weight from the left (or stationary weight)
- Window elements arrive already aligned so you do NOT need:
 - address generators
 - line buffer-based local reuse
 - muxing
 - padding logic

Your systolic array will reuse:

- pixel values as they propagate vertically
- weights as they propagate horizontally
- partial sums inside the array

The window generator ensures the inputs are already scheduled.

6 Final explanation (clean version)

The proposed window generator does **not** implement systolic reuse itself.

Instead, it **maximizes upstream data reuse** by maintaining row FIFOs and reducing redundant accesses, *and then outputs windows in a streaming pattern that perfectly matches the dataflow of a systolic array*. Therefore, it **enables and promotes reuse in the systolic array**, even though the reuse logic is not implemented inside the generator.

If you want, I can help you next with:

- ◆ Designing the systolic array interface that uses the window generator
- ◆ How to map each window column onto your PE array
- ◆ How many rows and columns your systolic array needs for best reuse
- ◆ Whether you should use weight-stationary, output-stationary, or row-stationary flow

Just tell me!



I am sending this email to my professor.
Can you please arrange this in a coherent way?

Good afternoon Professor,
I hope you are having a good day. Just to come back on my preoccupation on the systolic Array Architecture Design. When talking about this architecture to you, I didn't know that our final project will be on Convolution. In fact I am a second year PhD student with a pure software background and I started investing my learning in the FPGA 2 months before the beginning of the semester because I wanted to have some basis to cope with reconfig 1. I like your teaching patterns and I have register for reconfig2 next semester. and I want to direct my research on FPGA architecture for Robotic algorithm. And there are few robotic algorithm task that rely on sliding window operation. This is how I started jodding the 1st draft of my architecture last time.

I read the paper you shared during the lecture on FIFOs and buffer. And I still have to understand exactly how the Variable-Read FIFO works? How do we write data on the FIFO. I did understand why it's variable but I am not sure on how we know that we should feed the next

maxtrix row in the fifo.

And the way data is shifting in the coalester row register is not clear to me.

Also, I think that the architecture describe in this paper focuses on the window generation that feed the coalester in a great re-use format. However, from what I learnt on a systolic array data reuse aspect, most of the time a single pixel participates in multiple overlapping windows. If we assume a weight stationarry systolic array. I am willing to combine this work with the weight stationary systolic logic which comprise : PE to PE data propagation (it can be vertically, horizontally or both), partial sum accumlation. But I will start with a first clear version when it will be well set.

Please , do you have a couple of minute to discuss first about the architecture present on a paper and second a possible way to couple the reuse idea of the paper a systolic array logic.

Your help and support will matters a lot on how I will move forward into my project.

Here is a clear, professional, and well-structured version of your email.

I preserved your meaning, but fixed grammar, flow, and tone.

Subject: Clarification on Window Generator Architecture and Systolic Array Integration

Good afternoon Professor,

I hope you are having a great day. I wanted to follow up regarding my questions about the systolic array architecture we discussed earlier. At that time, I didn't know that our final project would focus on convolution. I also wanted to share a bit of context: I am currently a second-year PhD student with a purely software background, and I began learning FPGA concepts about two months before the semester started so I could prepare for Reconfigurable Computing 1.

I really appreciate your teaching style, and I have already registered for Reconfigurable Computing 2 next semester. My long-term goal is to direct my research toward FPGA architectures for robotic algorithms, and many robotics tasks rely on sliding-window operations. That is why I started drafting an initial systolic-array architecture when we talked last time.

I read the paper you shared during the lecture about FIFOs and buffers, but I still have a few points I need help understanding:

- I am still unsure how the **Variable-Read FIFO (VRF)** works in practice. I understand why the read size is variable, but I am not yet clear on how data is written into the VRF or how we know when to feed the next image row into the FIFO.
- I am also unsure about the way data shifts inside the **coalescer's row registers**. The internal movement of data within the coalescer is not completely clear to me yet.
- From what I understand, the architecture in the paper mainly focuses on efficient **window generation**, producing outputs in a format that supports data reuse very well. However, from the systolic-array perspective, especially with a **weight-stationary** dataflow, a single pixel is typically reused across multiple overlapping windows. I would like to explore how to combine the paper's window-generation approach with weight-stationary systolic concepts—such as PE-to-PE data propagation, partial-sum accumulation, and spatial reuse.

My plan is to first establish a clear, correct version of the window-generation architecture, and then explore how to couple it with a systolic dataflow.

If you have a few minutes sometime this week, I would greatly appreciate discussing (1) the architecture presented in the paper and (2) possible ways to combine its reuse strategy with systolic-array techniques.

Your help and guidance would be extremely valuable for the direction of my project.

Thank you very much, Professor.

If you want, I can also produce a **shorter** or more **formal** version.



hey Yin, these are the 2 resources I recommend you to get some basics knowledge in Verilog. To connect to the first link, you must use your UF account. The second link is a series of a YouTube videos on the basics of Verilog.

the way I recommend you to start is :

watch the first lectures of the youtube channel . Once you are done with that, connect to the first link and follow the entire training. It last 2 hours. Praticice all the challenge on your computer. Don't forget that the most important thing is to understand the core concepts, You can ping me anytime you are done and I will take a look on what you have done.

When following the course in the first link, from time to time I recommend you to watch the youtube channel classes in orther to feed your knowdlege in the tiny details thatt matters in verilog Programing.

feel free to ping me any time you are done witht the different challenge.

please polish this for me

Here is a polished and professional version of your message:

Hey Yin,

These are the two resources I recommend for getting a solid foundation in Verilog. For the first link, make sure to log in with your UF account. The second link is a YouTube playlist that covers the basics of Verilog in a very clear and beginner-friendly way.

Here's the order I suggest:

1. **Start with the first few lectures from the YouTube playlist** — they will give you a straightforward introduction to the basic concepts.
2. **After that, access the first link and complete the entire training** (it takes about 2 hours).

Make sure to practice all the challenges on your computer.

3. The most important thing is to focus on understanding the *core concepts*.

From time to time, I recommend going back to the YouTube videos while following the UF training — this will reinforce the little details that really matter in Verilog programming.

Feel free to ping me anytime you finish the challenges or if you have any questions. I'll be happy to take a look at your work.



Hi Kiara, my name is Audrey and I am Dr. Bobda PHD student. Our lab had a showcase last month and I had to print a poster at marston library. I do have a charges on my account due on december 1st. Dr. Bobda told me to share that with you and you will take care of it.

Thank you for your asistance,
Have a great rest of your day.

Best,

same prompt here

Here is a polished and professional version of your message:

Hi Kiara,

My name is Audrey, and I am Dr. Bobda's PhD student. Our lab had a showcase last month, and I printed a poster at Marston Library for the event. I now have a charge on my account that is due on December 1st. Dr. Bobda asked me to share this with you so that you could take care of it.

Thank you for your assistance.

Have a great rest of your day.

Best regards,

Audrey



what is the tittle of this email?

Here are a few appropriate subject line options you can choose from:

Best options:

- **Poster Printing Charge Assistance**
- **Charge on My Account for Lab Poster Printing**
- **Request for Assistance with Marston Poster Printing Charge**
- **Poster Printing Expense Submitted per Dr. Bobda**

If you want it short and simple:

- **Poster Printing Charge**

Let me know if you want it more formal or more concise!



Hey Yin,

These are the two resources I recommend for getting a solid foundation in Verilog. For the first link, make sure to log in with your UF account. The second link is a YouTube

playlist that covers the basics of Verilog in a very clear and beginner-friendly way.

Here's the order I suggest:

Start with the first few lectures from the YouTube playlist — they will give you a straightforward introduction to the basic concepts.

After that, access the first link and complete the entire training (it takes about 2 hours).

Make sure to practice all the challenges on your computer.

The most important thing is to focus on understanding the core concepts.

From time to time, I recommend going back to the YouTube videos while following the UF training — this will reinforce the little details that really matter in Verilog programming.

Feel free to ping me anytime you finish the challenges or if you have any questions. I'll be happy to take a look at your work.

suggest a tittle to this

Here are some good title options for your message:

Clear and Simple

- **Recommended Resources to Start Learning Verilog**
- **Getting Started with Verilog – Resources and Guidance**
- **Verilog Learning Plan and Resources**

More Supportive / Friendly

- **Verilog Beginner Guide: Resources + How to Start**
- **Your Verilog Learning Roadmap**
- **Introduction to Verilog: Recommended Resources**

Professional / Mentor Tone

- **Guided Learning Plan for Verilog Fundamentals**
- **Suggested Verilog Training Materials and Steps**

Let me know if you'd like it to sound more formal, casual, or technical!



can we design a memory with 3 output data?

Thought for 8s >

Short answer: **Yes** — you can design a memory with 3 read outputs, but how you do it depends on your FPGA, required performance, and area trade-offs. Below I explain the common approaches, pros/cons, and give small Verilog templates you can use or adapt.

Also — if you want me to relate this to the window-generator paper you gave earlier, here's the file you uploaded: `/mnt/data/fpga081-stitt.pdf`. I can read it and make suggestions that tie into your design.

Approaches to get 3 read outputs

1) Replicate the memory 3× (read-port replication) — easiest

- Instantiate **three copies** of the same RAM (or BRAM), drive every write into all three copies, and use each copy as one independent read port.
- Pros: single-cycle independent reads, very simple, easy to implement with FPGA BRAM primitives (which are usually dual-port or single-port).
- Cons: 3× area and BRAM usage; must update all copies on every write (coherence cost).

When to use: medium/large memories when timing and parallel reads matter and you can afford BRAMs.

2) Use true multi-ported RAM IP (multi-read multi-write) — if vendor IP available

- Some FPGA vendors or IP catalogs provide configurable multi-port RAMs (3+ read ports) using URAM/BRAM modes or tool-generated multiport IP.
- Pros: area-optimized versus naive replication, single-cycle reads.
- Cons: may be vendor-specific, sometimes limited depth/width, could be slow or resource-heavy under synthesis.

When to use: when you need true multi-port and vendor supports it.

3) Build a register-file for small memories — best for tiny RAMs

- Implement memory as an `reg [W-1:0] mem [0:DEPTH-1]`. This infers distributed RAM or flip-flop storage. You can create multiple read ports by returning values from the array combinatorially or synchronously.
- Pros: small depth -> very flexible, easy to give multiple read ports.
- Cons: not scalable to large depths (area blowup).

When to use: very small tables / FIFOs (tens to low hundreds of words).

4) Time-multiplex the reads (single BRAM + arbiter) — area-efficient but needs scheduling

- Use one BRAM and an arbiter/controller that sequences the three read addresses across different cycles (or different phases of a clock) and registers the outputs. If you can tolerate latency, you can give logically 3 outputs (over time), but not all valid simultaneously in one clock cycle.
- Pros: minimal BRAM usage.
- Cons: increased latency, lower throughput, added control complexity.

When to use: when you cannot afford replication and throughput/latency requirements allow scheduling.

5) Banked memory (interleaving) — trade single-cycle reads vs hashing

- Split memory into banks and map addresses to banks so that typical read patterns hit different banks. Can service multiple concurrent reads if they go to different banks. Needs careful banking scheme to avoid conflicts.
- Pros: better area than full replication, high throughput for well-behaved access patterns.
- Cons: bank conflicts require stalls or extra logic.

When to use: when access patterns are known and can be partitioned.

Practical recommendations (short)

- **Need three independent reads each cycle?** Replicate memory 3× or use vendor multi-port IP.
- **Memory is small (\leq few hundred words)?** Use register-file with 3 read ports.
- **Area-critical but read throughput can be lower or pipelined?** Time-multiplex a single BRAM.
- **Access pattern known and regular?** Bank the memory.

Example 1 — Replicated BRAM style (Verilog)

This uses three internal memories; a single write port writes all copies. All reads are synchronous (1-cycle latency typical for BRAMs).

verilog

 Copy code

```
module mem_3read #(
    parameter ADDR_WIDTH = 10,
    parameter DATA_WIDTH = 32
) (
    input wire          clk,
    input wire          we,
    input wire [ADDR_WIDTH-1:0] waddr,
    input wire [DATA_WIDTH-1:0] wdata,
    // three read ports
    input wire [ADDR_WIDTH-1:0] raddr0,
    output reg [DATA_WIDTH-1:0] rdata0,
    input wire [ADDR_WIDTH-1:0] raddr1,
    output reg [DATA_WIDTH-1:0] rdata1,
    input wire [ADDR_WIDTH-1:0] raddr2,
    output reg [DATA_WIDTH-1:0] rdata2
);

    // instantiate 3 memories (infers BRAMs or distributed RAMs)
    reg [DATA_WIDTH-1:0] mem0 [0:(1<<ADDR_WIDTH)-1];
    reg [DATA_WIDTH-1:0] mem1 [0:(1<<ADDR_WIDTH)-1];
    reg [DATA_WIDTH-1:0] mem2 [0:(1<<ADDR_WIDTH)-1];
```

```

// write to all three copies
always @(posedge clk) begin
    if (we) begin
        mem0[waddr] <= wdata;
        mem1[waddr] <= wdata;
        mem2[waddr] <= wdata;
    end
end

// synchronous read (1-cycle latency)
always @(posedge clk) begin
    rdata0 <= mem0[raddr0];
    rdata1 <= mem1[raddr1];
    rdata2 <= mem2[raddr2];
end

endmodule

```

Notes:

- Tools will likely infer three BRAMs (or distributed RAM) — check synthesis reports.
- If you need asynchronous reads, modify to combinational reads, but many BRAM primitives in FPGAs are synchronous.

Example 2 — Small register-file with 3 combinational read ports

If `DEPTH` is small, use `reg` array and return combinational outputs (zero-cycle read).

verilog

 Copy code

```

module regfile_3r #(
    parameter DEPTH = 64,
    parameter AW = 6,
    parameter DW = 32
)(
    input  wire          clk,
    input  wire          we,
    input  wire [AW-1:0] waddr,

```

```

    input  wire [DW-1:0]      wdata,
    input  wire [AW-1:0]      raddr0,
    input  wire [AW-1:0]      raddr1,
    input  wire [AW-1:0]      raddr2,
    output wire [DW-1:0]      rdata0,
    output wire [DW-1:0]      rdata1,
    output wire [DW-1:0]      rdata2
);

    reg [DW-1:0] mem [0:DEPTH-1];

    // write
    always @(posedge clk) begin
        if (we) mem[waddr] <= wdata;
    end

    // combinatorial read (inferred as LUTs/FFs for small depth)
    assign rdata0 = mem[raddr0];
    assign rdata1 = mem[raddr1];
    assign rdata2 = mem[raddr2];

endmodule

```

Be careful: very small depths only — synthesis will expand into LUTRAM or registers.

Example 3 — Time-multiplexed single BRAM (sketch)

If you can tolerate a few cycles latency and can schedule reads, use one BRAM and cycle through the three read addresses:

- On cycle N: present raddr0, read BRAM, store into reg0 next cycle.
- On cycle N+1: present raddr1, store into reg1.
- On cycle N+2: present raddr2, store into reg2.
- After 3 cycles you have all three outputs valid.

This requires an arbiter/round-robin FSM and read result registers. I can provide a template if you want this option.

Coherency & write ordering

- With replication, **writes must be forwarded to all copies in the same cycle** to keep copies consistent. If writes and reads can be simultaneous, you must define read-after-write behavior (e.g., write-through or write-first) and design accordingly.
- With time-multiplexing, if a write happens between read cycles, you must control ordering and possibly stall.

FPGA-specific notes

- Many FPGAs expose **true dual-port BRAM** (2 ports) easily. For 3 read ports you either replicate or use a vendor multi-port IP (if offered).
- For Xilinx: common pattern is to replicate BRAMs or use UltraRAM/BRAM cascades or vendor multi-port generator IP.
- For Intel/Altera: similar patterns; their block memory may be used with replication or multiport IP.

If you tell me:

1. **Target FPGA vendor/model** (Xilinx 7-series, Ultrascale, Intel Arria, etc.),
2. **Memory depth/width**, and
3. **Timing/throughput constraints** (do you need 3 independent reads every cycle? is latency OK?),

I'll give a concrete recommendation and provide the exact Verilog/BRAM instantiation or an IP-based approach tailored to your target.



which vendor give multi read RAM IP?

Most FPGA vendors allow **two-port RAM natively** (usually 1 write + 1 read, or 2 read/write ports), but **true multi-read (>2 read ports) RAM** is *not* directly supported in hardware.

However, some vendors provide **IP generators** that *emulate* multi-port RAMs using replication, banking, or time-multiplexing.

Here is the complete list:

✓ 1. Intel (Altera) – YES, supports Multi-Port RAM via IP

Intel's *Platform Designer (Qsys)* and *Megafunction IP Catalog* include:

📌 "On-Chip Memory (RAM or ROM)" IP

- Supports multi-port configurations
- Can create **multi-read, single-write** memories
- Internally uses replication + muxing
- You can choose:
 - 1 write port
 - Up to N read ports (configurable)

Vendor: Intel FPGA

Devices: Cyclone, Arria, Stratix families

Tool: Quartus Prime

This is one of the easiest ways to get 3+ read ports **without coding the replication yourself**.

✗ 2. Xilinx – NO direct multi-read RAM IP

Xilinx does **not** provide a native 3-read-port RAM generator.

Their BRAM/URAM primitives support:

- Single-port
- Simple dual-port
- True dual-port (2R, 2W, or 1R1W)

For 3+ read ports, you must use:

✓ Memory replication

✓ Block RAM duplications

✓ Register-file built from LUTRAM

BUT:

Xilinx *XPM_MEMORY* primitives do **not** support >2 ports.

So for Xilinx:

- **You must manually replicate BRAMs** for 3 read ports
- Or create a **distributed RAM register file** with multiple combinational read ports
- Or use **HLS** (Vivado HLS) which lets you synthesize multi-port arrays using internal replication

Vendor: Xilinx

Devices: 7-series, UltraScale, Versal

Tools: Vivado, Vitis HLS

3. Lattice – No true multi-port RAM IP

Lattice FPGAs (iCE40, ECP5, CrossLink, etc.) have:

- Single-port RAM
- Dual-port RAM

But **no multi-port RAM generator**.

If you need 3 read ports:

- You replicate RAM blocks
- Or use LUT-based memory

Vendor: Lattice

Tools: Radiant, Diamond

4. Microchip (Microsemi) – No multi-port RAM IP

Microchip's SmartFusion/PolarFire families support:

- Single-port
- Dual-port

No 3+ port IP.

You must manually replicate memory.

Summary Table

Vendor	Native Multi-Port IP (>2 read ports)	Notes
Intel/Altera	✓ Yes	Best support. Multi-read IP available.
Xilinx	✗ No	Must replicate BRAM or use LUTRAM.
Lattice	✗ No	Must replicate memory manually.
Microchip	✗ No	Same—only dual-port.

 So the only major vendor that gives TRUE multi-read RAM IP is:

★ Intel (Altera)

via their:

 “On-Chip Memory (RAM or ROM)” multi-port IP

 Generated in Quartus

This supports **1 write + N parallel reads** (as many as you configure, limited by area).

If you tell me your FPGA vendor (Xilinx or Intel),

I can show you the *exact* recommended method for building 3-read-port RAM in that architecture, including:

- ✓ BRAM replication diagrams
- ✓ Verilog templates
- ✓ Timing analysis
- ✓ Area trade-offs

Just tell me the device family!



does xilinx support dual read memory RAM IP?

Yes — **Xilinx *does* support dual-read (2-read-port) RAM**, but only through the **True Dual-Port (TDP) Block RAM mode**, not through a special “multi-read” IP block.

Here’s the breakdown:

✓ Xilinx supports 2-read-port RAM using True Dual Port BRAM

Xilinx Block RAM (BRAM) primitives (RAMB18, RAMB36, URAM) all support:

✓ True Dual Port (TDP) mode

Each port can independently be configured as:

- Read
- Write
- Read/Write

So you can configure it as:

Port A = Read

Port B = Read

Which gives you a **dual-read RAM**.

This **is supported in Vivado** through:

1) Block Memory Generator IP

You select:

- **Memory Type:** True Dual Port RAM
- Set both ports as **read-only**

2) XPM_MEMORY_TDPRAM

Xilinx parameterizable memory (recommended for RTL designs)

It supports:

- **2 independent read ports (if you configure both ports as read)**
- Or 1 read + 1 write
- Or 2 read/write

What Xilinx does *NOT* support

- No built-in IP for **3 or more** read ports
- No native block RAM mode with >2 ports
- You must replicate memory for more ports

Example: Using Block Memory Generator for Dual-Read

When configuring the IP:

- Interface Type: **Native**
- Memory Type: **True Dual Port RAM**
- Port A: **Read**
- Port B: **Read**

This gives you a single BRAM with two read outputs.

Example using XPM in Verilog

verilog

 Copy code

```

xpm_memory_tdpram #(
    .ADDR_WIDTH_A(10),
    .ADDR_WIDTH_B(10),
    .MEMORY_SIZE(1024*32),
    .MEMORY_PRIMITIVE("block"),
    .READ_LATENCY_A(1),
    .READ_LATENCY_B(1),
    .WRITE_DATA_WIDTH_A(32),
    .READ_DATA_WIDTH_B(32)
) my_dual_read_ram (
    .clka(clk),
    .addra(raddr0),
    .douta(rdata0),
    .ena(1'b1),

    .clkb(clk),
    .addrb(raddr1),
    .doutb(rdata1),
    .enb(1'b1),

    .wea(0), // no writes on port A
    .web(0)
);

```

This gives you **two independent synchronous read ports** with a single BRAM block.

Summary

✓ Xilinx supports dual-read (2-read-port) RAM

via two methods:

- **Block Memory Generator IP → True Dual Port Mode**
- **xpm_memory_tdpram**

✗ Xilinx does not support >2 read ports

(extra ports require memory replication).

If you want, I can show you how to:

- Configure the Block Memory Generator for dual-read
- Write a proper dual-read RTL wrapper
- Build a 3-read or 4-read memory using BRAM replication

Just tell me what size/depth you need!



in which scenario do we need that kind of ram?

Dual-read (2-read-port) RAM is extremely useful in many hardware architectures because it lets **two different data elements be read in the same clock cycle** without stalling the pipeline.

Here are the main scenarios where you *specifically need* **true dual-read RAM**:

1. Sliding Window / Convolution Engines

This is directly related to your research.

When generating windows (e.g., for a 3×3 or 5×5 kernel), you often need to fetch **multiple pixels from different rows of memory at the same time**.

Example:

- Port A reads pixel from row i
- Port B reads pixel from row $i+1$

This avoids:

- Multi-cycle stalls
- Pipeline bubbles
- Extra buffering

 **Convolution cores and window generators frequently use dual-read RAM.**

✓ 2. Register Files in CPUs / GPUs (2R1W)

Most processor register files need:

- **2 read ports**
- **1 write port**

This enables instructions like `ADD x = a + b`, where the processor must read `a` and `b` *in the same cycle*.

Xilinx BRAMs help implement:

- RISC-V register files
- DSP blocks
- Processing engines

➡ **Any design that needs to read two operands at once uses dual-read RAM.**

✓ 3. Systolic Arrays with Weight-Stationary or Output-Stationary Dataflow

In systolic arrays, each Processing Element (PE) may require:

- current input activation
- neighbor activation
- weight vectors

Often, two operands must be fetched per cycle:

- Port A → feed PE row
- Port B → feed PE column

➡ **Dual-read RAM allows simultaneous feeding of systolic PEs without stalling.**

✓ 4. State Machines Needing Two Lookups Per Cycle

Some finite state machines (FSMs) or controllers use lookup tables (LUT-based memories) to retrieve:

- next-state
- output vector

These two reads must happen simultaneously.

✓ 5. Dual-Buffering / Ping-Pong Buffers

When data is stored and processed in parallel, a dual-read RAM allows:

- reading two independent buffers
 - cross-fading buffers
 - scanning two addresses each cycle (e.g., interpolation, pixel pairing, compression)
-

✓ 6. Cryptography engines (AES, DES, SHA)

Crypto cores often need:

- two S-box lookups
- two key schedule reads
in the same cycle.

Dual-read RAM reduces latency.

✓ 7. Machine Learning Layers with Parallel Data Paths

Layers like:

- BatchNorm
- LayerNorm
- Depthwise convolution
- Max-pooling

often require multiple reads per cycle to sustain throughput.

🧠 Why is dual-read so important?

If you have only **one read port**, you must read the two values in **two cycles**, which:


- cuts maximum throughput in half
- adds extra pipeline stages
- increases control logic complexity

- slows down systolic arrays or window generators

Dual-read RAM solves this cleanly.

Summary — When do we need dual-read RAM?

Scenario	Why it's needed
Sliding window generators	Fetch multiple pixels from different rows at once
Convolution engines	Maintain throughput without stalls
Systolic arrays	Feed multiple operands per PE cycle
CPU/GPU register files	2 operands read per instruction
Crypto pipelines	Parallel S-box/lookups
FSMs with large LUTs	Multiple reads per tick
DSP / FIR filters	Access two taps simultaneously

 If you want, I can also explain whether your *specific systolic array architecture* needs dual-read RAM, depending on your dataflow:

- weight-stationary
- input-stationary
- output-stationary
- row-stationary

Just tell me your architecture sketch and I'll map the memory port requirements.



in this scenary, we assumes that data is already written in the ram?

Great question — **yes**, in almost all dual-read RAM use-cases, we assume the data has already been written into the RAM *before* the dual-read phase begins.

But let me break it down clearly, because this is extremely important for architecture design.

General Rule

Dual-read RAM is used when you need to read two data values *in the same cycle*. This assumes the RAM has already been pre-loaded with valid data.

This is typical in:

- Convolution window generators
- Systolic arrays
- Register files
- DSP filters
- Crypto engines
- Any pipelined architecture

More precise explanation

In hardware pipelines, operations usually follow this order:

1. Load phase (write data into RAM)

- Data comes from DMA, AXI stream, FIFO, or a camera feed
- The RAM is filled row by row, or block by block
- This phase may take many cycles

2. Compute phase (dual-read mode)

- Now the RAM is *read* repeatedly and predictably
- Often the write port is idle (not used)
- The dual-read RAM provides 2 pixels/weights/activations per cycle

3. (Optional) Overlap read/write if using TDP RAM

Xilinx True Dual Port RAM allows:

- Port A = write
 - Port B = read
- Simultaneously (if addresses do not collide)

But **dual-read (R-R)** mode assumes:

- Write is complete
- Memory contains a full buffer/image/tile

Let's apply this to your sliding window / systolic array scenario

✓ RAM is first filled with image pixels (from DMA FIFO)

This is the **input buffer**.

✓ Once the buffer is ready, your window generator will:

- Read pixel Row(i) from Port A
 - Read pixel Row(i+1) from Port B
- in the same cycle**

This is how you build:

- Line buffers
- Coalescers
- Window buffers
- Column shift registers

✓ The systolic array also assumes data is ready

Weights are preloaded (weight-stationary)

Activations are streamed and reused across cycles

Partial sums travel along PEs

! BUT: Some architectures *do* write and read at the same time

Depending on the design flow:

Scenario A — Preload then compute

- Write phase (fill memory)
- Compute phase (dual-read only)

This is simplest.

Scenario B — Streaming with ping-pong buffers

- Buffer 0 is being read by the systolic array
- Buffer 1 is being filled by DMA

After finishing:

- Swap roles (ping ↔ pong)

This maintains continuous throughput.

Scenario C — Xilinx TDP RAM with 1 write + 1 read

Technically:

- Port A = Write
 - Port B = Read
- at the same time

But this is NOT dual-read — it's 1R1W.

To get 2 reads + 1 write, you would need:

- 1 dual-read memory (replicated BRAM)
- 1 write port feeding both replicated memories

Final Answer

Yes — dual-read RAM assumes that the memory already contains valid data.

Data is written **before** dual-read use, or via a separate port during the process (not common for strict R-R mode).

Dual-read RAM is designed for:

- reading two values simultaneously
- from a *pre-filled* buffer

If you want, I can draw a diagram showing:

- write phase
- read phase
- systolic array feed
- buffer structure
- how many RAMs you need

Just tell me!



with the paper I shared with you on data re-use, do we have any situation where we stall in the pipeline?
Does it always deliver output at each clock cycle?

Great question — and the answer is **subtle**, because the window-generation architecture in the paper *looks* fully pipelined, but it actually has **specific conditions where it can stall**, and it does **not** always produce a valid output every clock cycle.

Let's break it down clearly based on the architecture described in your paper **fpga081-stitt.pdf** (The Coalescing Window Architecture).

✓ 1. Does the architecture ever stall?

✓ **Yes — the architecture *can* stall in several situations.**

Even though it is designed to maximize data reuse and apply pipelining, stalls can occur in these cases:

(A) When the FIFO line buffers become empty

The architecture uses **Variable-Read FIFOs** to buffer incoming rows.

A stall happens if:

- The coalescer requests the next row
- But the corresponding FIFO does not yet have enough pixels

This produces a **read stall** until more data arrives.

👉 Especially likely at image boundaries or when multiple rows are consumed faster than data is written.

(B) During row transitions

When sliding from one window row to the next window row, the controller relies on the FIFO delivering the next row.

If the new row is not ready, the pipeline stalls.

(C) At startup (pipeline fill time)

Just like any pipelined system, the first window cannot be produced until:

- Line buffers are filled with **(window_height – 1)** rows
- The coalescer shift registers have valid data

During this initial phase → **no output**.

(D) Handling dynamic window sizes in the 10×10 → 3×3 case

When the max window is 10×10 but user requests a smaller window:

- The architecture still slides a *full 10×10 area*
- Some elements are zero padded
- But the control logic must ensure the physical image row is available

Padding doesn't cause stalling, but the striping effect at boundaries **can**.

Conclusion for stalls

► **Yes — there are situations where the pipeline stalls.**

The architecture is *not* always throughput-constant.

2. Does the system always produce an output every clock cycle?

✓ **No — it does *not* produce a valid output every cycle across the whole execution.**

It produces one output per cycle **only after**:

1. The line buffers are full
2. The coalescer is fully loaded
3. The controller has stable data from all row FIFOs
4. No FIFO starvation occurs

This condition is called:

steady-state streaming

Only in steady-state does the architecture achieve:

- **1 window per clock cycle**
- **Maximum throughput**
- **No stall cycles**

But not before that, and not always at boundaries.

3. When does the design produce one window per cycle?

The architecture produces **1 output per clock** during:

- Middle of each row
- Middle of each column
- When FIFO buffers contain all required rows

- When the stride is 1 (classic overlapping windows)

This is the optimal throughput period.

4. When does output NOT occur every cycle?

- ✗ At image borders
- ✗ At beginning of frame
- ✗ At end of frame
- ✗ When sliding past image boundaries (zero padding period)
- ✗ When row FIFOs haven't reached required fill level
- ✗ When window size \leq max window (controller steps cause internal gaps)

So "one output per cycle" is **not globally true**, only within a fully filled window pipeline.

★ Final Summary

Question	Answer
Does the architecture ever stall?	✓ Yes, especially if FIFOs are empty or during row transitions.
Does it always output every cycle?	✗ No — only in steady state. Initial fill and boundary cases reduce throughput.

If you want, I can create:

- ✓ A pipeline timing diagram (with fill, steady, drain phases)
- ✓ A cycle-accurate stall analysis table
- ✓ A comparison vs your systolic array reuse plan
- ✓ When to replicate RAM vs when to use multiple FIFOs

Just tell me!



is it worthy to have a pipeline that deliver valid output every cycle for N time and then when we reach a new row, we have W (windows number of rows) time we it delivers invalid output. And N and W values are constant. It never changes.

Thought for 5s >

Short answer: **Yes — it can be perfectly worthwhile**, but whether it's *the best* choice depends on the utilization you get ($N/(N+W)$), how much hardware/energy you can afford, and whether you can hide or reuse the “invalid” cycles. Below I explain the trade-offs, give concrete formulas and examples, and list practical alternatives and next steps you should try.

What your scheme is

You propose a pipeline that:

- Produces **valid outputs for N consecutive cycles** (steady-state),
- Then, at a row boundary, produces **W consecutive invalid outputs** (dead cycles),
- N and W are constant and predictable.

So one steady block of work, then a fixed dead period, repeating.

Key metric: utilization

Define utilization U as fraction of cycles that are useful:

$$U = \frac{N}{N + W}$$

- If $U \approx 1$ (e.g. 0.9+), the wasted cycles are small and the design is attractive.
- If U is low (e.g. 0.5), you're burning half your clock cycles — costly in energy and wasted bandwidth.

Example: $N=240, W=16 \rightarrow U = 240/256 = 0.9375 \rightarrow$ excellent.

$N=8, W=7 \rightarrow U = 8/15 \approx 0.533 \rightarrow$ probably not acceptable.

Pros of your approach

1. **Simplicity & predictability.** Constant N and W makes controller FSMs simple, easier timing closure, and easier worst-case analysis.
2. **High peak throughput.** During N cycles you can fully saturate PEs/memory bandwidth.
3. **Low control overhead.** No dynamic arbitration, no complex bank conflict resolution. Good for FPGA timing.
4. **Good fit with the paper's window generator.** The `window_gen` produces long steady runs and then border effects — mapping those runs to N/W blocks is natural. See your uploaded paper: `/mnt/data/fpga081-stitt.pdf`.

Cons / costs

1. **Wasted cycles = wasted bandwidth & energy.** Memory and PEs idle during W cycles.
2. **Hardware cost vs. throughput trade-off.** To avoid W you'd need more buffering, banking, or replication (area cost).
3. **Latency and jitter.** Outputs arrive in bursts; downstream logic must tolerate that (or use FIFOs).
4. **Not ideal for small N .** If N is small relative to W , overall throughput and efficiency suffer.

When this design *is* a good choice

- $N \gg W$, i.e. utilization high ($> \sim 0.8$).
- Your application tolerates bursty outputs (or you have FIFOs downstream).

- You prefer simpler control logic and smaller routing/fanout (favors timing closure).
- Memory bandwidth is plentiful during the N window and can be idled during W, or you can do useful background work in W cycles (see below).

Ways to make it better (hide or reuse the W cycles)

1. **Ping-pong / double-buffering:** While pipeline produces output from buffer A (the N cycles), refill buffer B in the W cycles. Swap roles next block. This hides writes and can eliminate stalls if memory/host DMA can keep up.
2. **Interleave work from multiple images/tiles:** Instead of letting pipeline sit idle, feed it windows from other tiles/images during W cycles — effectively multiplex workloads to keep hardware busy and raise utilization.
3. **Banking / address mapping:** Reorganize memory so row transitions don't cause large W. E.g., interleave rows across banks so reads at boundary hit other banks.
4. **Replication:** Replicate buffers so different reads don't conflict — removes the need for dead cycles at the cost of BRAM/URAM.
5. **Prefetch & compute overlap:** Use W cycles to do auxiliary work (weight loads, partial-sum writes, reduction) so nothing is idle.
6. **Increase N by larger tile sizes:** Process bigger tile so W becomes small fraction. But that may increase buffer size.

Bandwidth & PE sizing (practical check)

If your systolic array needs R input values per cycle to keep PEs busy during N cycles, required **peak** memory bandwidth is $B_{peak} = R \times f_{clk}$. Because you only need that peak for fraction U of the time, **average** required bandwidth is:

$$B_{avg} = B_{peak} \times U = B_{peak} \times \frac{N}{N + W}$$

But note: memory and DMA hardware typically need to sustain B_{peak} during N cycles. So ensure DRAM/PCIe/DMA can deliver that bursty traffic.

Concrete checklist to decide if it's worthy

1. Compute $U = N/(N + W)$. Is $U \geq 0.8$? Good. $U < 0.6$? Reconsider.
2. Can your memory subsystem deliver the **burst** B_{peak} required during N cycles? If not, you'll stall inside N and the model breaks.
3. Can downstream logic accept bursty outputs or can you add a FIFO to smooth them?
4. Do you have resources to implement banking/ping-pong/replication if needed?
5. Can you hide W cycles by overlapping with DMA writes, prefetch, or other useful tasks?

Recommendation / suggested path

1. **Measure N and W for your real parameters.** Use image size, window height H, and p (parallelism) to compute N and W exactly.
2. **Compute utilization U** and effective throughput. If $U \geq 0.8$, proceed with simple design.
3. **Simulate** cycle-accurate: model your VRF/window buffer timings from the paper (see /mnt/data/fpga081-stitt.pdf) and confirm steady-state and boundary behavior.
4. If U is low, try one of: ping-pong buffering, interleaving multiple tiles, or banking. Re-evaluate area vs. throughput.
5. If you want, I can:
 - compute N and W from your image/window parameters,
 - produce a utilization and bandwidth table, or
 - sketch an interleaving/ping-pong controller to hide W.

If you share your concrete numbers (image width/height, window height H, p, desired output-per-second, memory bandwidth available), I'll compute N , W , U , B_{peak} , and show whether the simple constant N/W pipeline is a good practical design for your target.



for a matrix 32×32 with a windows of 3×3 , how many sliding windows left to right do we have , righth to left and top to bottom?.

the number of sliding windows to reach anew row is when we move from top to bottom. AND for each move we have 3 invalid outputs.

The other sliding moving direction(left to right and right to left) gives us valid output.

is it worth it in this scenario?

Thought for 20s >

Good question — let's compute it step by step and then judge whether the pattern is worth it for your $32 \times 32 / 3 \times 3$ case.

Counts (stride = 1, 3×3 window on a 32×32 image)

- Horizontal positions (left \rightarrow right) per row = $32 - 3 + 1 = 30$ windows.
So each left \rightarrow right sweep of a row yields **30 valid windows**.
The right \rightarrow left sweep is the same: **30 valid windows**.
- Vertical positions (top \rightarrow bottom) = $32 - 3 + 1 = 30$ distinct row positions.
- Total windows overall = $30 \times 30 = 900$ sliding windows.

Your pipeline pattern

You said: produce N valid outputs while scanning left \leftrightarrow right (here $N = 30$), and when you move down to the next row you have $w = 3$ invalid cycles (window-height = 3). That implies:

- Number of row transitions = number of row moves = $\text{rows} - 1 = 30 - 1 = 29$.
- Invalid cycles total = $29 \text{ transitions} \times 3 \text{ invalid cycles} = 87 \text{ invalid cycles}$.

So total cycles (valid + invalid) = $900 + 87 = 987$.

Utilization U = valid cycles / total cycles = $900/987 \approx 0.91185 \rightarrow \approx \mathbf{91.19\%}$.

(Computed from the uploaded paper/notes you shared: /mnt/data/fpga081-stitt.pdf.)

Short summary of numbers

- Valid windows per row ($L \rightarrow R$ or $R \rightarrow L$): **30**
- Total window positions (all rows): **900**
- Number of row transitions: **29**
- Invalid cycles per transition: **3**
- Total invalid cycles: **87**
- Total cycles = **987**
- Utilization $\approx \mathbf{91.2\%}$

Is it worth it?

Yes — in this scenario it is usually worthwhile.

Why:

- A utilization of **~91%** is high: only ~9% of cycles are wasted.
- You get high *peak* throughput during the valid-window bursts (30 consecutive valid outputs), which is good for keeping PEs busy in a systolic array.
- The control logic is simple and predictable (constant N and W), which helps timing closure on FPGA.

Caveats / things to check:

1. **Burst bandwidth:** Memory/DMA must deliver the peak data rate during the N valid cycles. If the memory can't supply the burst, you'll get extra stalls inside N and the effective utilization will drop.
2. **Downstream buffering:** If downstream stages can't accept bursty outputs, you'll need a FIFO to smooth the stream (adds area and latency).
3. **Startup/drain latency:** There's pipeline fill latency before the first valid window and drain after the last; account for it if you process many small images.
4. **Resource trade-off:** Eliminating the W invalid cycles (e.g., by more buffering, banking, or replication) costs BRAM/FF/logic. Given $U \approx 91\%$, that extra hardware

might not be worth the modest gain.

Practical suggestions

- If memory can sustain bursts and downstream logic tolerates bursts → keep the simple N/W scheme (it's efficient and easy).
- If you must push utilization higher, try:
 - **Ping-pong buffering** (fill alternate buffer during W cycles), or
 - **Interleaving** tiles/images so other work fills W cycles, or
 - **Banking** to avoid row-boundary stalls (if access pattern permits).
- I can compute the exact required peak bandwidth (B_{peak}) and show whether your DRAM/PCIe/DMA can meet it if you tell me the data width, clock frequency and p (parallel windows per cycle).

Would you like me to compute B_{peak} and show a concrete memory-bandwidth check for your design (give me clock rate, bytes-per-pixel, and parallelism p)?



Don't compute based on the upload paper I shared. Compute in a logical systolic array idea where we have 3 rows FIFO connected to each row of the systolic array. each FIFO is connect to the RAM. the row FIFO size is equal to the windows column size. for the first cycles, we feed all the row FIFO(3 column per row in our case.). once it is done, we send data to the systolic array PE that implement PE-PE horizontal data reuse. for the first windows, each time we have a valid input, a new input is inserted on the row FIFO and the output data(which leave the systolic array) is store in another FIFO buffer for reuse (we have 2 FIFO buffer for use corresponding to the 1st and second Row). Then when we slide from top to down, the first TOP PE lose their and we transfer the data from the bottom PE to the The top PE (at this time we have invalid data). to slide from the right to the left, we will use the 3 reuse before we created before. After the first left to right, top to bottom and right to

left, we now need only one row fifo(the 3rd row fifo) to keep fetching in the RAM.

Thought for 20s >

Nice — thanks for the clear description of your systolic-flow. I'll compute the cycle counts, utilization, and give a short judgement and practical recommendations. I'll *not* use the uploaded paper's numbers for the calculation (per your request), but I'll mention its path for reference: `/mnt/data/fpga081-stitt.pdf`.

Assumptions (from your description)

- Image: **32 × 32** pixels.
- Window: **3 × 3** (width = 3, height = 3).
- Stride = 1.
- You have **3 row-FIFOs**, each FIFO depth = window column size = **3** (so each row-FIFO holds 3 pixels).
- **Startup**: you first fill all three row-FIFOs (i.e., load 3 elements per row × 3 rows).
- **Left → Right (L → R)** and **Right → Left (R → L)** passes produce **valid** outputs every cycle once FIFOs are loaded.
- **Top → Bottom (row transition)** causes **W = 3 invalid cycles** (you said when sliding down you get 3 invalid outputs).
- After the first L → R, T → B, R → L sweep, you then only need one row-FIFO (3rd) to keep fetching from RAM — but we'll count the initial full-fill cost anyway.

Step-by-step counts

1) Windows per row

$$\text{windows_per_row} = 32 - 3 + 1 = 30$$

So each horizontal sweep (L → R or R → L) yields **30 valid outputs**.

2) Total valid windows (all rows)

$$\text{rows_positions} = 32 - 3 + 1 = 30$$

$$\text{total_valid} = 30 \times 30 = 900$$

3) Row transitions (where invalid cycles occur)

You move from one row-position to the next 30 times total row-positions, so transitions
 $= 30 - 1 = 29$.

Per transition you stated **W = 3 invalid cycles**, so

$$\text{total_invalid_between_rows} = 29 \times 3 = 87$$

4) Startup fill (initially filling all 3 row-FIFOs)

Each FIFO holds 3 elements; filling 3 FIFOs requires:

$$\text{startup_cycles} = 3 \text{ rows} \times 3 \text{ elements} = 9$$

(After these 9 cycles the first valid window can be produced.)

5) Totals and utilization

$$\text{total_cycles} = \text{startup_cycles} + \text{total_valid} + \text{total_invalid_between_rows} = 9 +$$

Utilization U (fraction of cycles that are valid outputs):

$$U = \frac{900}{996} \approx 0.9036 (\approx 90.36\%)$$

Interpretation — is it worth it?

- **Utilization $\approx 90.4\%$** is high. Only $\approx 9.6\%$ of cycles are “wasted” (startup + row-boundary invalids). For a streaming systolic array this is generally very attractive: high peak throughput, simple control, good PE utilization during the 30-cycle steady runs.
- The design *does* buy important data-reuse benefits: