Parallel Programming and Heterogeneous Computing

FPGA Accelerators

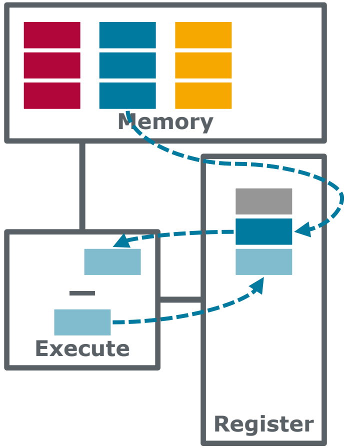
Max Plauth, Sven Köhler, Felix Eberhardt, *Lukas Wenzel* and Andreas Polze

Operating Systems and Middleware Group

Example:

Given Arrays a, b, and f calculate r[i] = a[i] × f[i] + b[i] × (1 - f[i])

```
 LD  R0, #0
loop:
 LD  R1, [f + R0]
 SUB R2, #1, R1
 LD  R3, [a + R0]
 LD  R4, [b + R0]
 MUL R5, R3, R1
 MUL R6, R4, R2
 ADD R5, R5, R6
 ST  [r + R0], R5
 ADD R0, R0, #1
 BLT R0, #N, loop
```



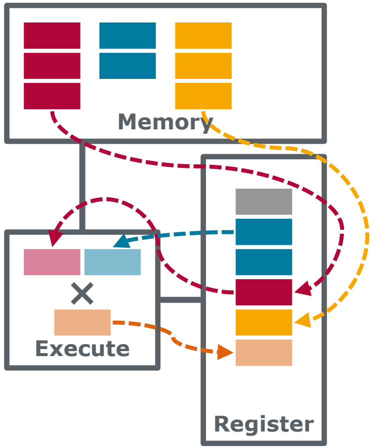
General Purpose Hardware **Custom Hardware**

Introduction
Mapping Workloads to Hardware

Example:

Given Arrays a, b, and f calculate r[i] = a[i] × f[i] + b[i] × (1 - f[i])

```
 LD  R0, #0
loop:
 LD  R1, [f + R0]
 SUB R2, #1, R1
 LD  R3, [a + R0]
 LD  R4, [b + R0]
 MUL R5, R3, R1
 MUL R6, R4, R2
 ADD R5, R5, R6
 ST  [r + R0], R5
 ADD R0, R0, #1
 BLT R0, #N, loop
```



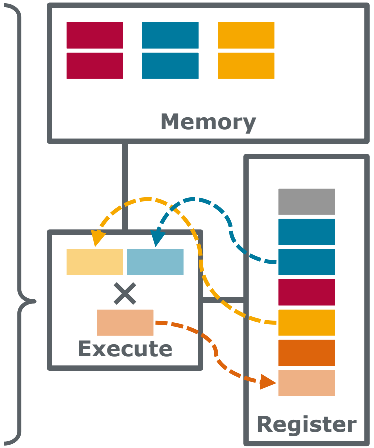
General Purpose Hardware

Custom Hardware

Example:

Given Arrays a, b, and f calculate r[i] = a[i] × f[i] + b[i] × (1 - f[i])

```
 LD  R0, #0
loop:
 LD  R1, [f + R0]
 SUB R2, #1, R1
 LD  R3, [a + R0]
 LD  R4, [b + R0]
 MUL R5, R3, R1
 MUL R6, R4, R2
 ADD R5, R5, R6
 ST  [r + R0], R5
 ADD R0, R0, #1
 BLT R0, #N, loop
```



General Purpose Hardware

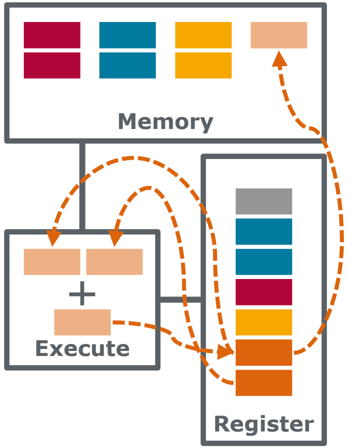
Custom Hardware

Mapping Workloads to Hardware

Example:

Given Arrays a, b, and f calculate r[i] = a[i] × f[i] + b[i] × (1 - f[i])

```
 LD  R0, #0
loop:
 LD  R1, [f + R0]
 SUB R2, #1, R1
 LD  R3, [a + R0]
 LD  R4, [b + R0]
 MUL R5, R3, R1
 MUL R6, R4, R2
 ADD R5, R5, R6
 ST  [r + R0], R5
 ADD R0, R0, #1
 BLT R0, #N, loop
```



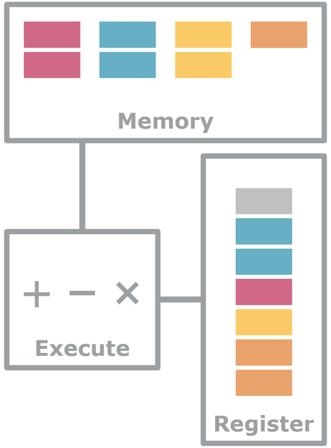
General Purpose Hardware **Custom Hardware**

Introduction
Mapping Workloads to Hardware

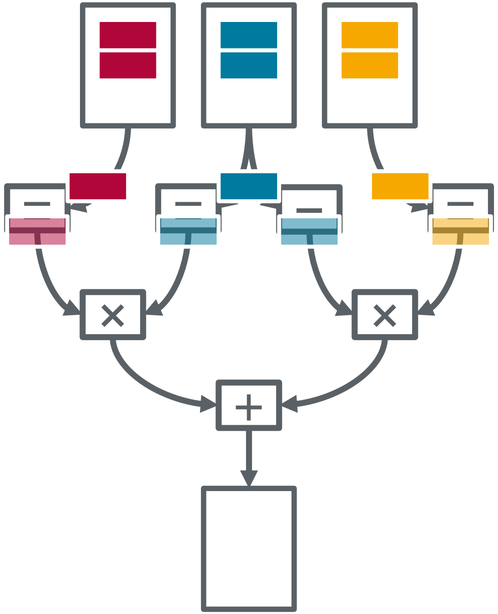
Example:

Given Arrays a, b, and f calculate `r[i] = a[i] × f[i] + b[i] × (1 - f[i])`

```
 LD  R0, #0
loop:
 LD  R1, [f + R0]
 SUB R2, #1, R1
 LD  R3, [a + R0]
 LD  R4, [b + R0]
 MUL R5, R3, R1
 MUL R6, R4, R2
 ADD R5, R5, R6
 ST  [r + R0], R5
 ADD R0, R0, #1
 BLT R0, #N, loop
```



General Purpose Hardware



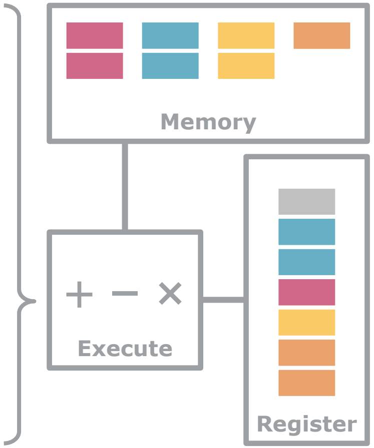
Custom Hardware

Introduction
Mapping Workloads to Hardware

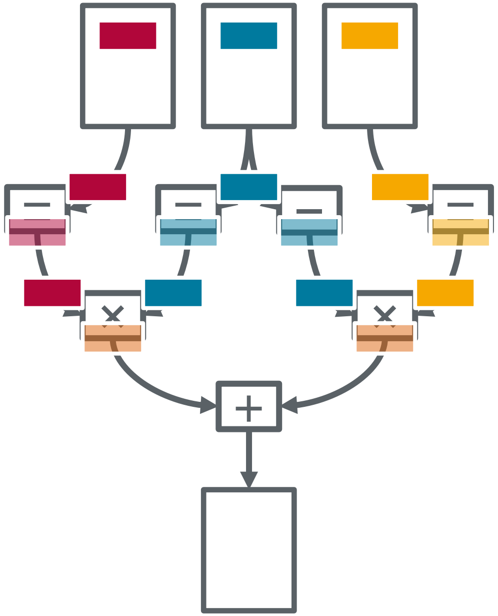
Example:

Given Arrays a, b, and f calculate `r[i] = a[i] × f[i] + b[i] × (1 - f[i])`

```
 LD  R0, #0
loop:
 LD  R1, [f + R0]
 SUB R2, #1, R1
 LD  R3, [a + R0]
 LD  R4, [b + R0]
 MUL R5, R3, R1
 MUL R6, R4, R2
 ADD R5, R5, R6
 ST  [r + R0], R5
 ADD R0, R0, #1
 BLT R0, #N, loop
```



General Purpose Hardware



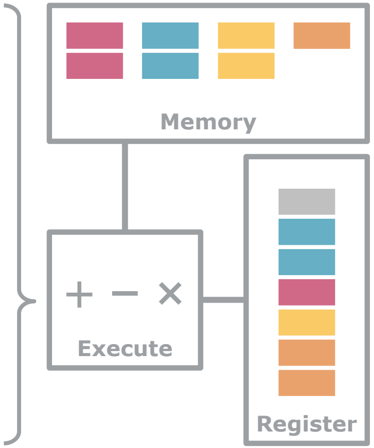
Custom Hardware

Introduction
Mapping Workloads to Hardware

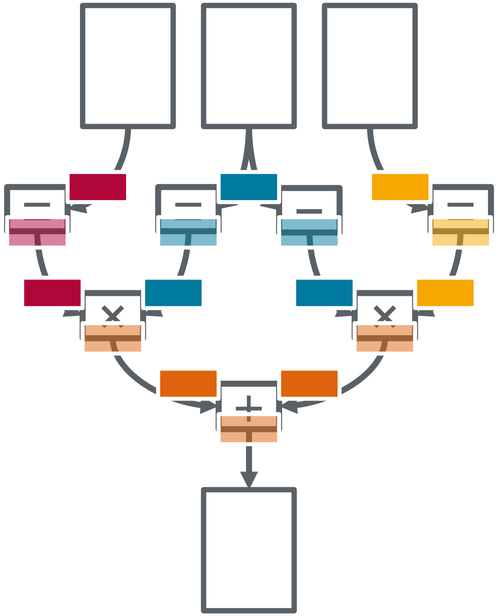
Example:

Given Arrays a, b, and f calculate r[i] = a[i] × f[i] + b[i] × (1 - f[i])

```
 LD  R0, #0
loop:
 LD  R1, [f + R0]
 SUB R2, #1, R1
 LD  R3, [a + R0]
 LD  R4, [b + R0]
 MUL R5, R3, R1
 MUL R6, R4, R2
 ADD R5, R5, R6
 ST  [r + R0], R5
 ADD R0, R0, #1
 BLT R0, #N, loop
```



General Purpose Hardware



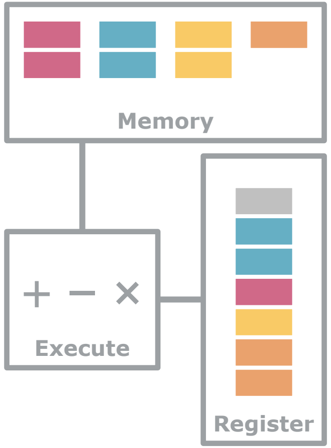
Custom Hardware

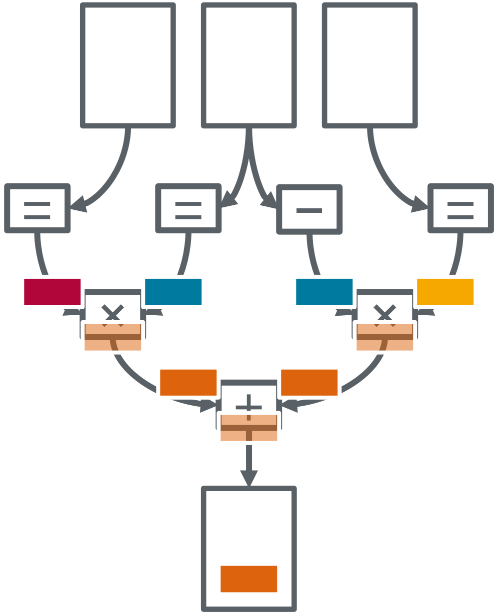
Introduction
Mapping Workloads to Hardware

Example:

Given Arrays a, b, and f calculate r[i] = a[i] × f[i] + b[i] × (1 - f[i])

```
 LD  R0, #0
loop:
 LD  R1, [f + R0]
 SUB R2, #1, R1
 LD  R3, [a + R0]
 LD  R4, [b + R0]
 MUL R5, R3, R1
 MUL R6, R4, R2
 ADD R5, R5, R6
 ST  [r + R0], R5
 ADD R0, R0, #1
 BLT R0, #N, loop
```





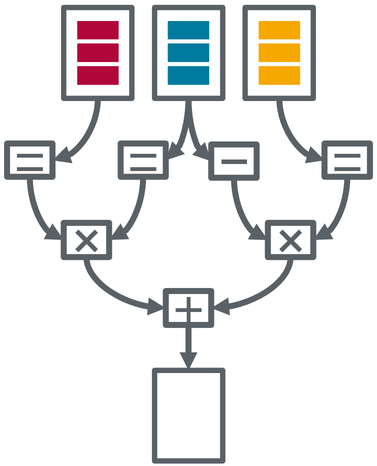
General Purpose Hardware

Custom Hardware

Introduction
Mapping Workloads to Hardware

- Truly custom hardware built as Application-Specific Integrated Circuits (ASICs) is extremely expensive to design and manufacture
  - Only feasible for high production volumes
  - Usually requires at least some general-purpose aspects to fit many use-cases

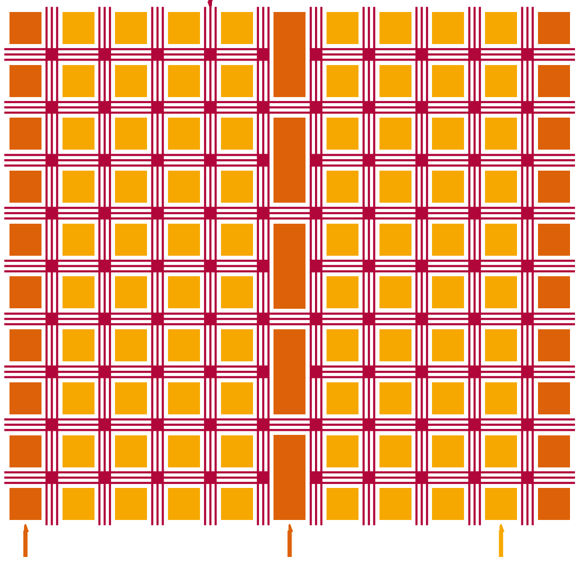
- Field Programmable Gate Arrays (FPGAs) are manufactured as general-purpose integrated circuits, and thus far less expensive than equivalent ASICs
- FPGAs can be configured to realize a custom hardware architecture



FPGA Characteristics
Hardware Structure

- Regular fixed-function integrated circuits implement a single and usually highly optimized hardware architecture (e.g. CPUs, GPUs, …)

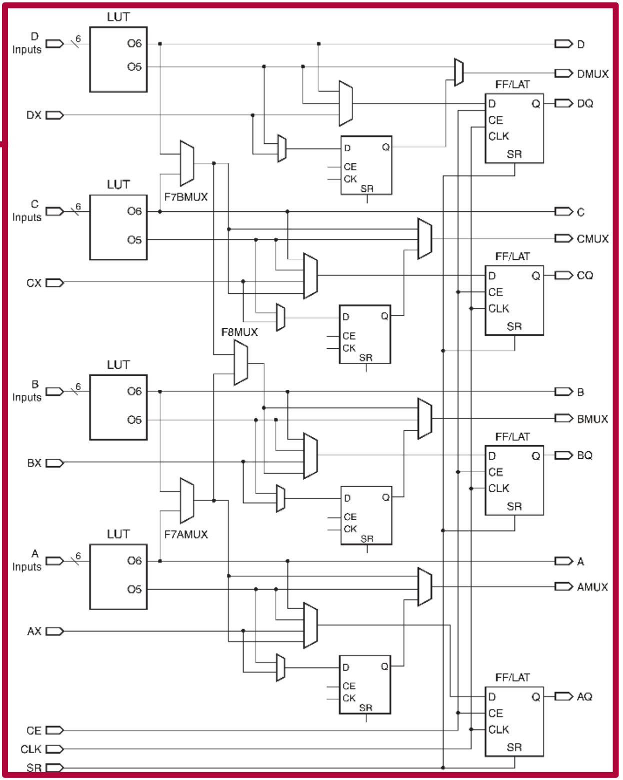
- FPGA fabric is a regular structure of hardware primitives and an interconnect for signal lines
  - Interconnect can be configured to connect signals lines between primitives
  - Primitives can be configured to select variations of their basic behavior
- ➢ **Appropriate configurations can make the FPGA behave like any custom hardware design (within fabric capacity)**



FPGA Characteristics
Hardware Structure

Hardware primitives include:

- Logic Blocks (CLB) with Flipflops, Lookup Tables, Multiplexers, …
- Memory Blocks (BRAM) to act as single port, dual port or FIFO memories
- Arithmetic Blocks (DSP) with hardware multipliers, adders, shifters, …
- Clock Management Blocks (MMCM) to derive clock signals with specific frequency and phase relations
- IO Banks with logic for various signaling standards

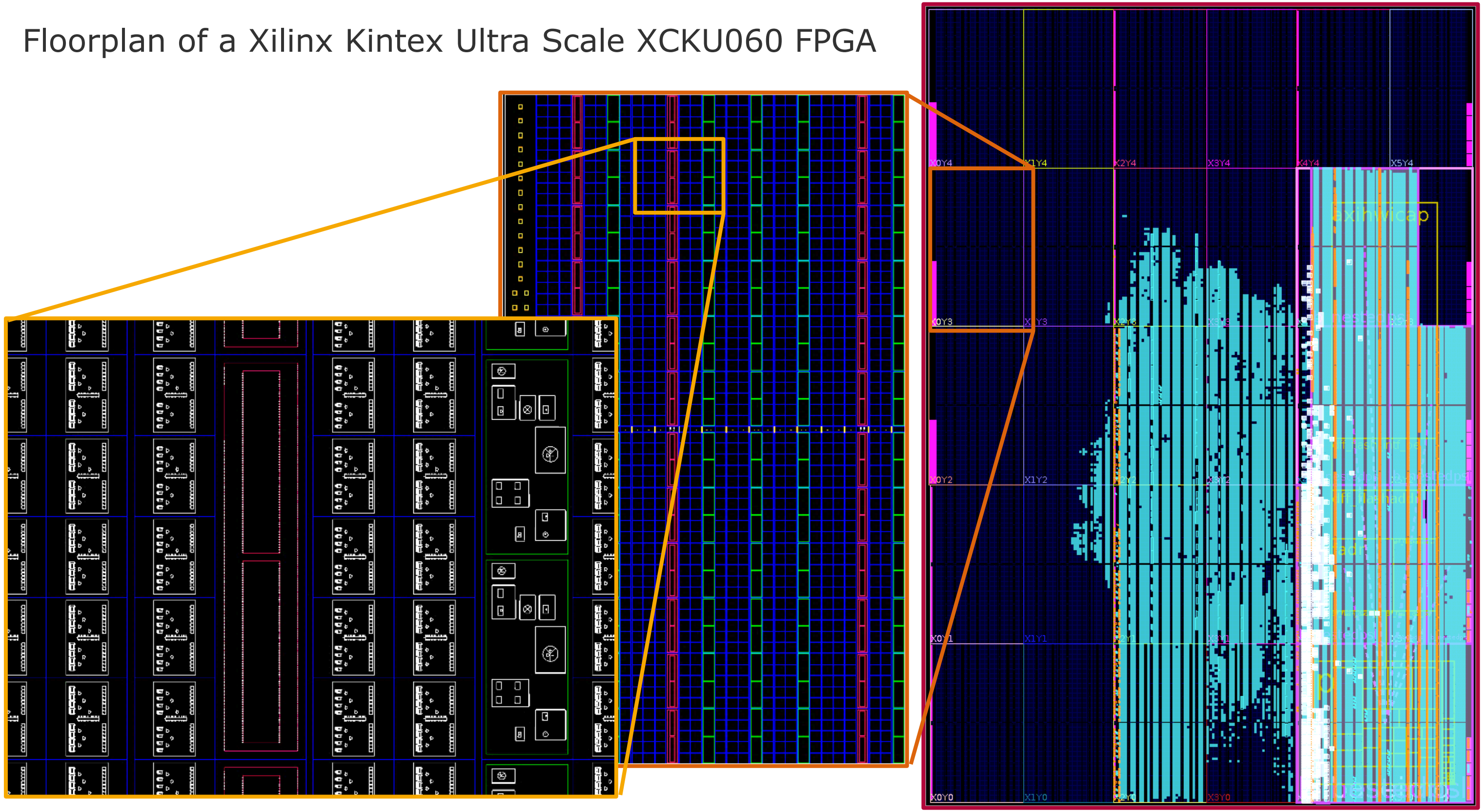


CLB in a Xilinx UltraScale FPGA (from: Xilinx UG 474, Figure 5-1)

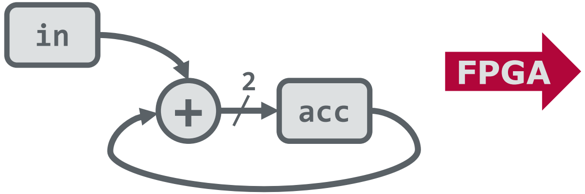
FPGA Characteristics
Hardware Structure

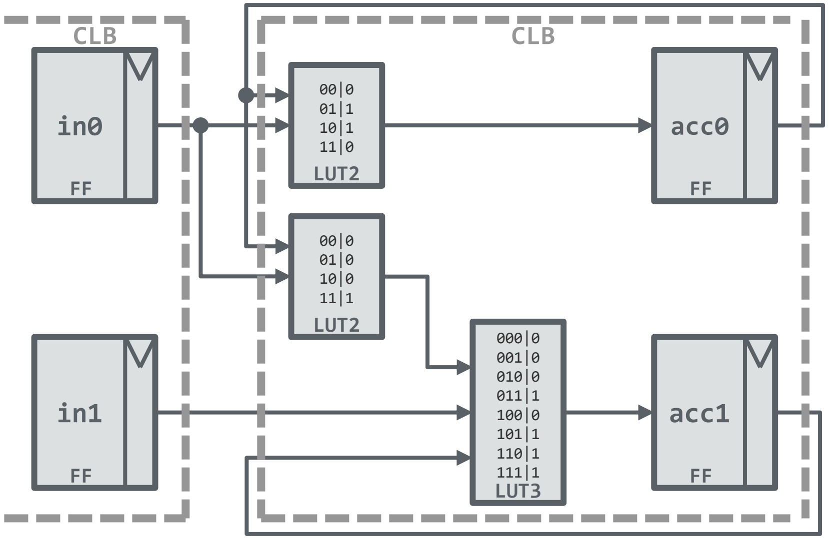
Hasso Plattner Institut

Floorplan of a Xilinx Kintex Ultra Scale XCKU060 FPGA



Example: Accumulator (2 bit)





FPGA Characteristics
Performance

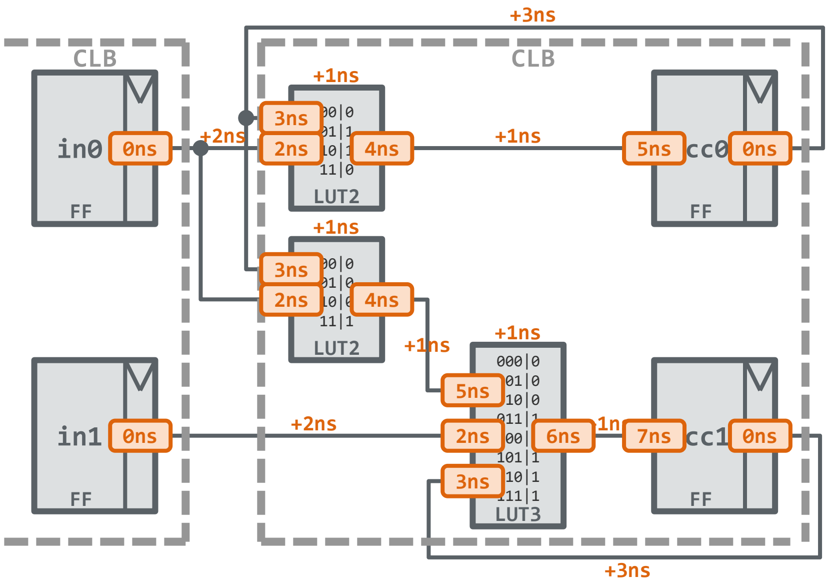
- Fixed-function hardware is rated by maximum operating clock frequency
- FPGAs have no uniform clock frequency rating:
  - FPGA fabric supports multiple clock signals in different regions
  - Specific configurations define combinatorial paths of varying lengths
  - ➢ Maximum **clock frequency is design specific** and constrained by the longest combinatorial path delay

- Specific primitives like BRAMs can have maximum clock frequency ratings
  - BRAMs on current Xilinx FPGAs run at up to 800MHz
- Individual logic delays range from 0.1ns to 0.5ns
  - ➢ Small and tightly coupled design sections may run at 1GHz
- Common frequency for complete designs is <u>250MHz</u>

FPGA Characteristics Performance

Example: Accumulator (2 bit)

- Combinatorial paths begin and end at flipflops
- Clock period must be longer that the maximum path delay



Maximum delay:

$$\max\{t_\delta\} = 7\text{ns}$$

Clock frequency:

$$f \leq \frac{1}{\max\{t_\delta\}} = 143\text{MHz}$$

FPGA Characteristics
Performance

FPGA designs operate at up to an order of magnitude lower clock frequencies than ASIC accelerators!

How do FPGAs achieve speedups over fixed function hardware?

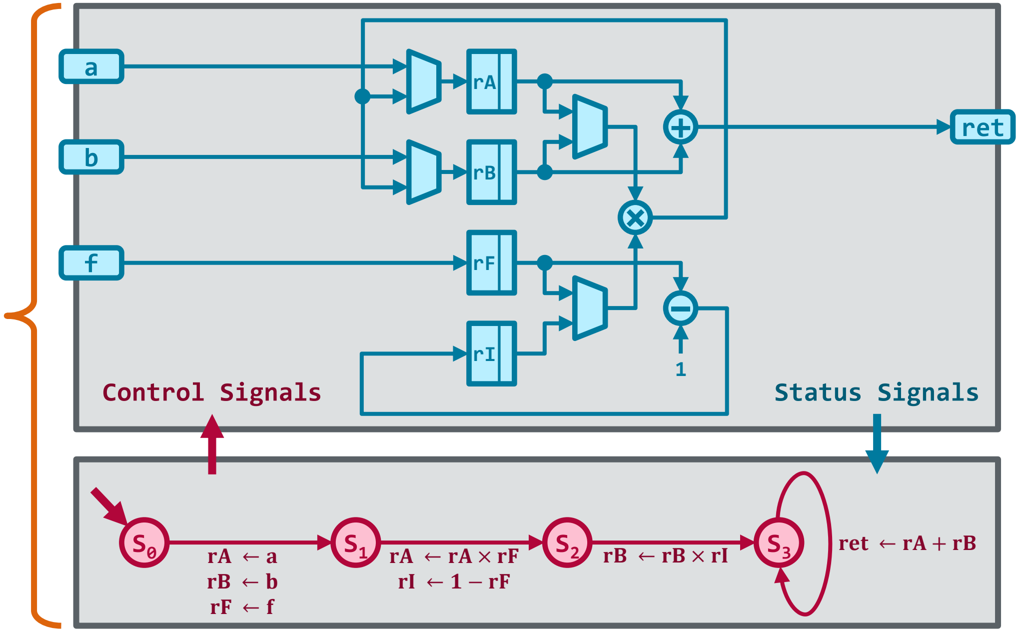
- **Avoid overheads** of general-purpose hardware:
  - CPUs invest a large amount of logic and cycles into **fetching and decoding general-purpose instructions**
  - CPUs must accommodate a wide variety of applications by providing a **compromise set of execution facilities** (i.e. function units, forwarding paths, …)

FPGA Design
Basic Patterns

Any program can be transformed into an equivalent hardware design:

- Variables and operations are realized in the **datapath**
- Control flow is realized through a **finite state machine** (FSM) controlling the datapath

```
int proc(int a, int b, int f)
{
  int f_inv = 1 - f;
  a *= f;
  b *= f_inv;
  return a + b;
}
```



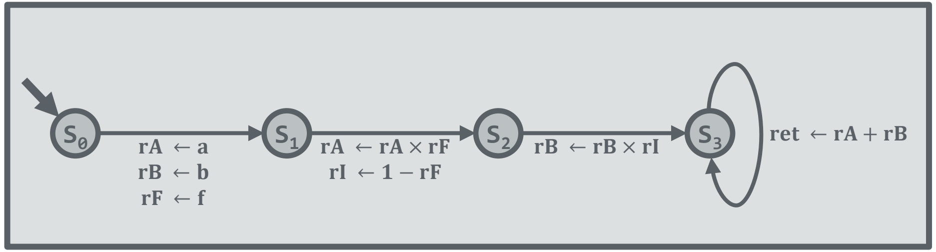
FPGA Design
Basic Patterns

Strictly reproducing the original control flow always yields a correct hardware implementation for a program.

- ! **Resulting design is rarely efficient**, as original control flow is ignorant of datapath utilization and does not capture data dependencies
- ➢ Efficient designs leverage **pipelining** and **replication** of operations to maximize computational throughput

```
int proc(int a, int b, int f)
{
  int f_inv = 1 - f;
  a *= f;
  b *= f_inv;
  return a + b;
}
```

=

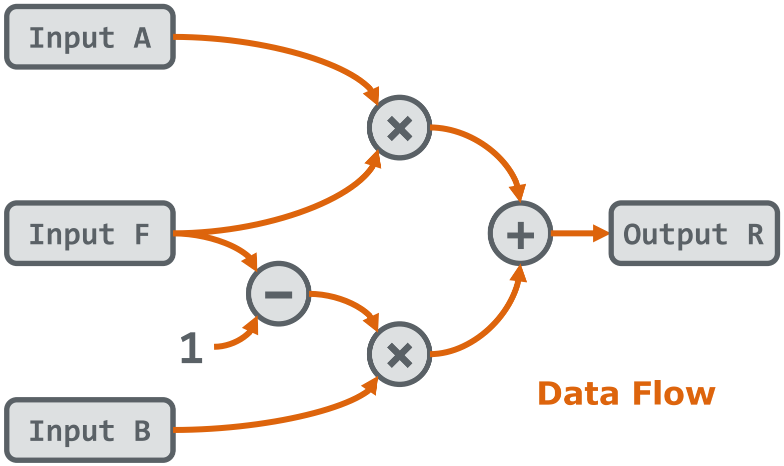


FPGA Design
Dataflow Model

- Dataflow is a computational model based on **streams of data units**, that are processed by traversing a **network of operators**
  - ➢ Enables a **flexible kind of task parallelism**, where operations are not orchestrated by control flow but availability of data operands

```
int proc(int a, int b, int f)
{
  int f_inv = 1 - f;
  a *= f;
  b *= f_inv;
  return a + b;
}
```

Control Flow



Data Flow

➢ **Workloads with an efficient dataflow representation usually yield an efficient hardware implementation!**

HDLs share syntactic features with programming languages:

- VHDL is related to Ada, Verilog to C

HDLs have fundamentally different semantics to programming languages:

- **Statements** are not executed in sequential order,
  but applied **concurrently**, whenever their input values change

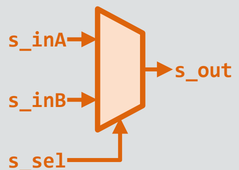
- **Function calls** have no meaning,
  closest equivalent are **module instantiations**,
  that like inline functions copy the module to the place of instantiation

Each (synthesizable) HDL construct translates to specific hardware structures:

- Conditional Statements → **Multiplexer**
- Signals that change value only on clock events → **Flipflops**
- Arithmetic operations → **Adder circuits**, **DSP Blocks**
- Reading and writing large arrays → **Distributed RAM**, **BRAM**

```
process (s_sel)
begin
  if s_sel = '0' then
    s_out <= s_inA;
  else
    s_out <= s_inB;
  end if;
end process;
```

=



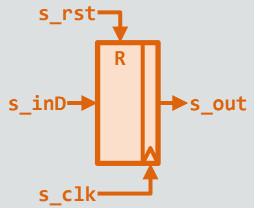
➢ **Designers need to know relations between HDL and hardware constructs to produce correct and efficient designs**

Each (synthesizable) HDL construct translates to specific hardware structures:

- Conditional Statements → **Multiplexer**
- Signals that change value only on clock events → **Flipflops**
- Arithmetic operations → **Adder circuits**, **DSP Blocks**
- Reading and writing large arrays → **Distributed RAM**, **BRAM**

```
process (s_clk)
begin
  if s_clk'event and s_clk = '1' then
    if s_rst = '1' then
      s_out <= '0';
    else
      s_out <= s_inD;
    end if;
  end if;
end process;
```

=



➢ **Designers need to know relations between HDL and hardware constructs to produce correct and efficient designs**

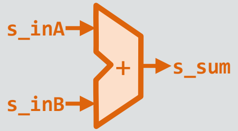
FPGA Development
Hardware Description Languages

Each (synthesizable) HDL construct translates to specific hardware structures:

- Conditional Statements → **Multiplexer**
- Signals that change value only on clock events → **Flipflops**
- Arithmetic operations → **Adder circuits**, **DSP Blocks**
- Reading and writing large arrays → **Distributed RAM**, **BRAM**

```
process (s_inA, s_inB)
begin
  s_sum <= s_inA + s_inB;
end process;
```

=



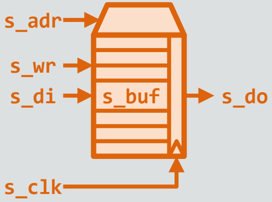
➢ **Designers need to know relations between HDL and hardware constructs to produce correct and efficient designs**

Each (synthesizable) HDL construct translates to specific hardware structures:

- Conditional Statements → **Multiplexer**
- Signals that change value only on clock events → **Flipflops**
- Arithmetic operations → **Adder circuits**, **DSP Blocks**
- Reading and writing large arrays → **Distributed RAM, BRAM**

```
process (s_clk)
begin
  if s_clk'event and s_clk = '1' then
    if s_wr = '1' then
      s_buf(to_integer(s_adr)) <= s_di;
    end if;
    s_do <= s_buf(to_integer(s_adr));
  end if;
end process;
```

=

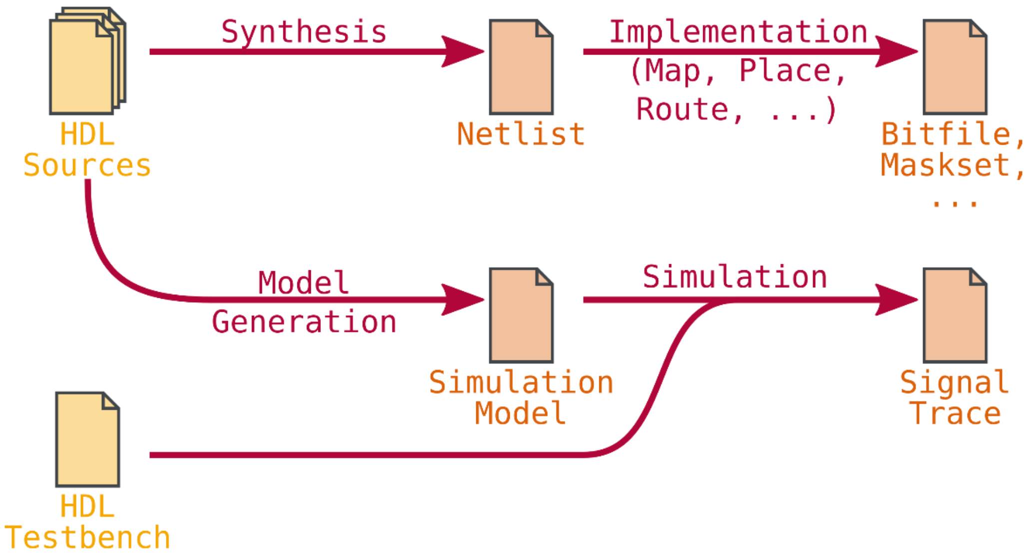


➢ **Designers need to know relations between HDL and hardware constructs to produce correct and efficient designs**

FPGA Development Workflow

Hardware development toolchains and workflows are significantly different from software development.

Final artifacts are not executable binaries but hardware configurations.



FPGA Development
High-Level Design Methods

HDLs operate at a very low level of abstraction:

- HDL development requires **rare skillset** in developers as well as **much time and effort**

➢ Increase productivity by **raising level of abstraction** of design method

High-Level Synthesis (HLS):

- Automatically translate **programs** (usually restricted subset of C/C++) **into** equivalent **hardware descriptions**

+ No transition from software mindset + Well suited for algorithmic specification	– No fine-grained control over hardware – Not suited for structural specification

Block Designs (BD):

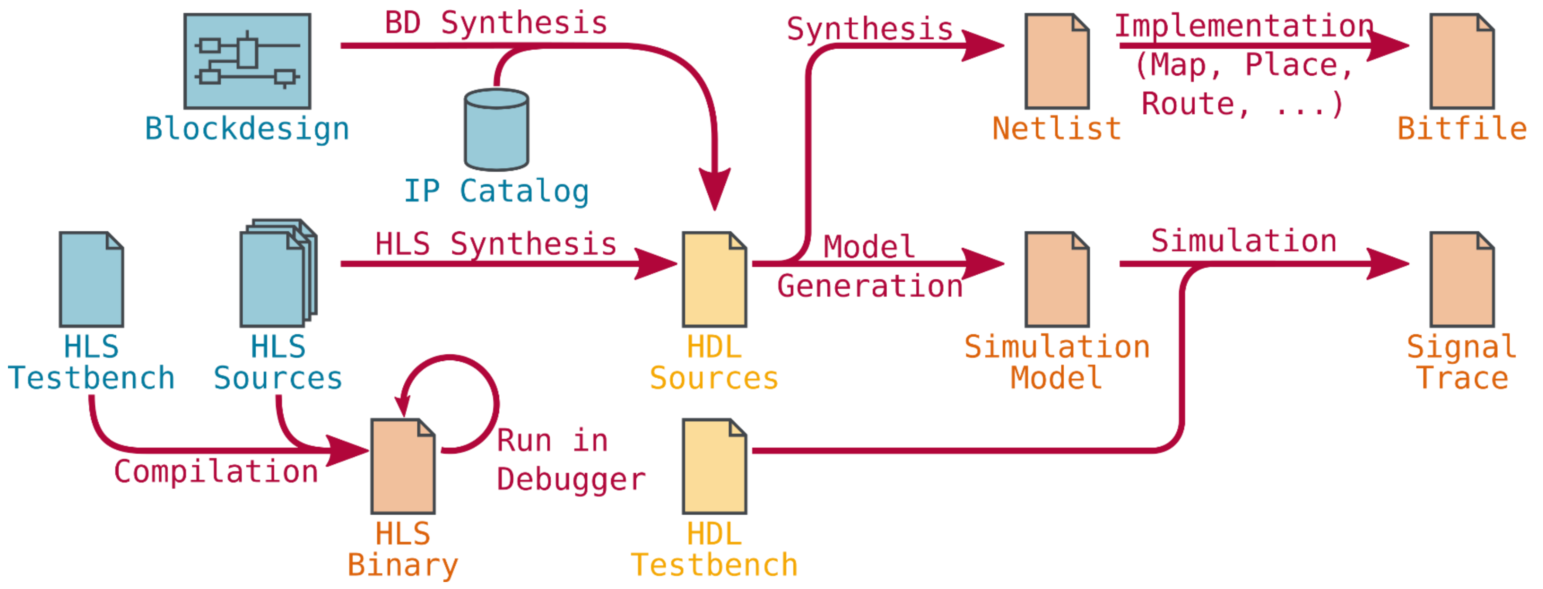
- **Instantiate** and connect **existing hardware modules** in a block diagram editor

+ Intuitive graphical method + Well suited for structural specification	– Relies on already defined modules – Not suited for algorithmic specification

FPGA Development Workflow

High-level design methods extend the frontend of traditional workflows.

They usually produce HDL descriptions as intermediate artifacts.



And now for a break and a bowl of Bancha.

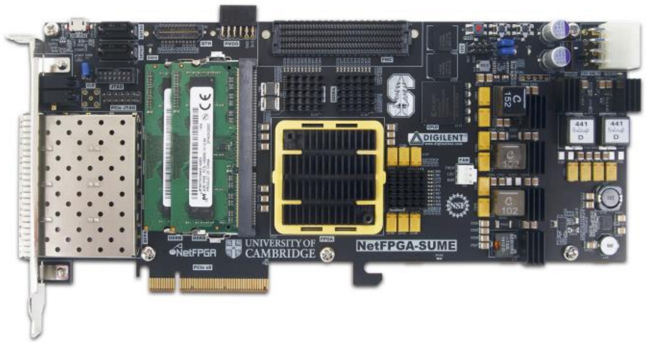
*or beverage of your choice

FPGA Accelerators

FPGA accelerator cards provide a **host system interface** as well as **local memory and IO resources**.

- DRAM modules to complement the limited BRAM capacity on the FPGA
- Flash Storage
- Network Interfaces
- Video and Peripheral Ports
- Auxilliary Accelerators like Crypto Units or A/V Codecs

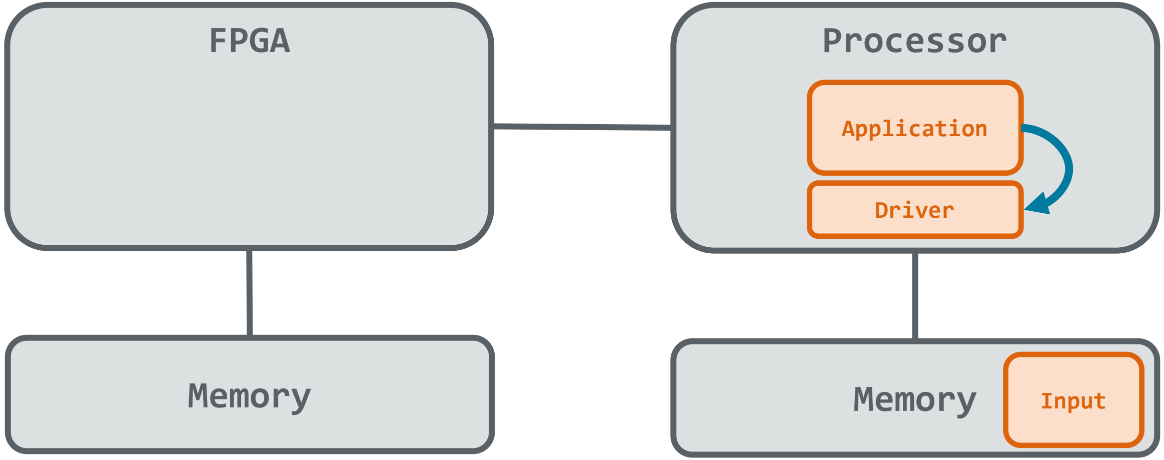
…



FPGA Accelerators

Device Attached Accelerators:

- Accelerator acts as a device in host system
- Accelerator can only access local resources
  - Host must copy data via DMA

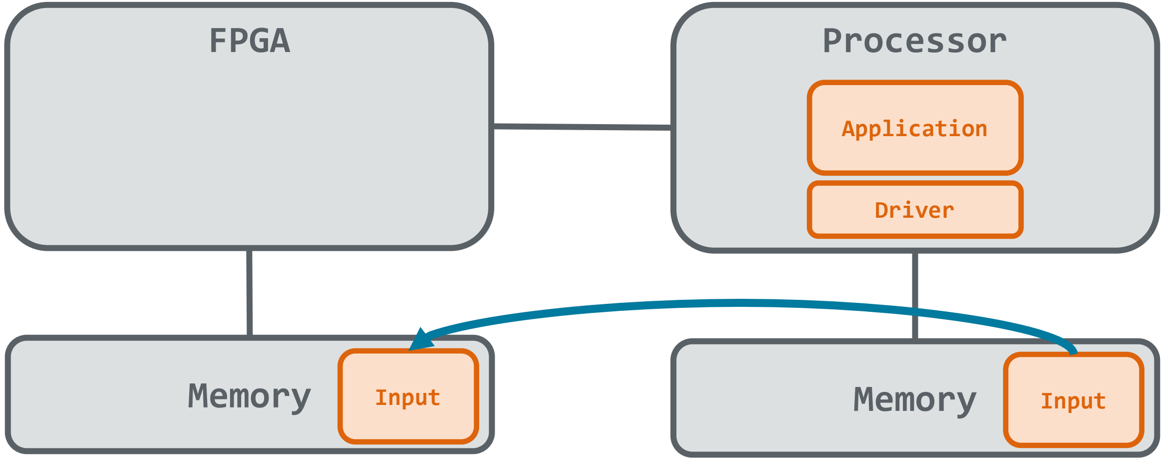


1. **Initiate**
2. **Copy**
3. **Process**
4. **Copy**
5. **Complete**

FPGA Accelerators

Device Attached Accelerators:

- Accelerator acts as a device in host system
- Accelerator can only access local resources
  - Host must copy data via DMA

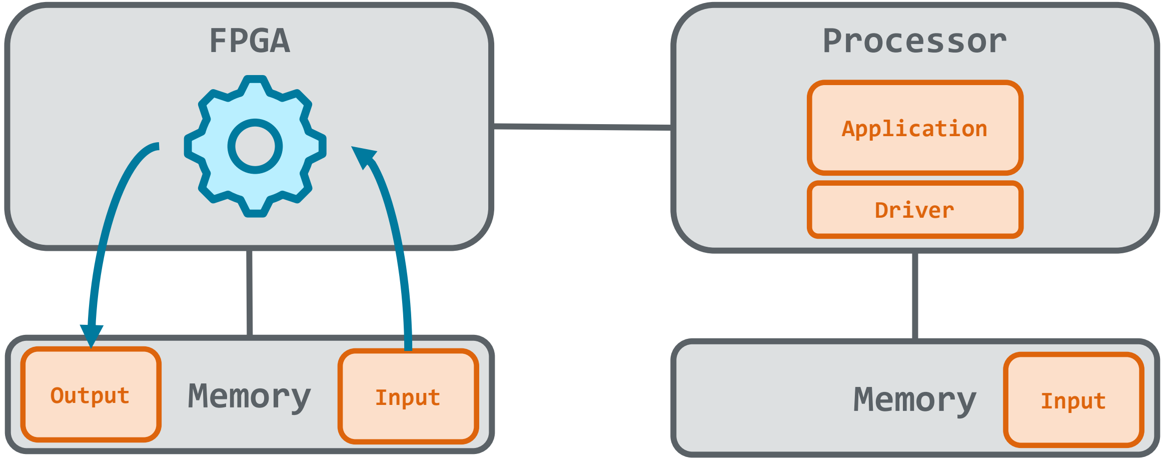


1. Initiate
2. **Copy**
3. Process
4. Copy
5. Complete

FPGA Accelerators

Device Attached Accelerators:

- Accelerator acts as a device in host system
- Accelerator can only access local resources
  - ➢ Host must copy data via DMA

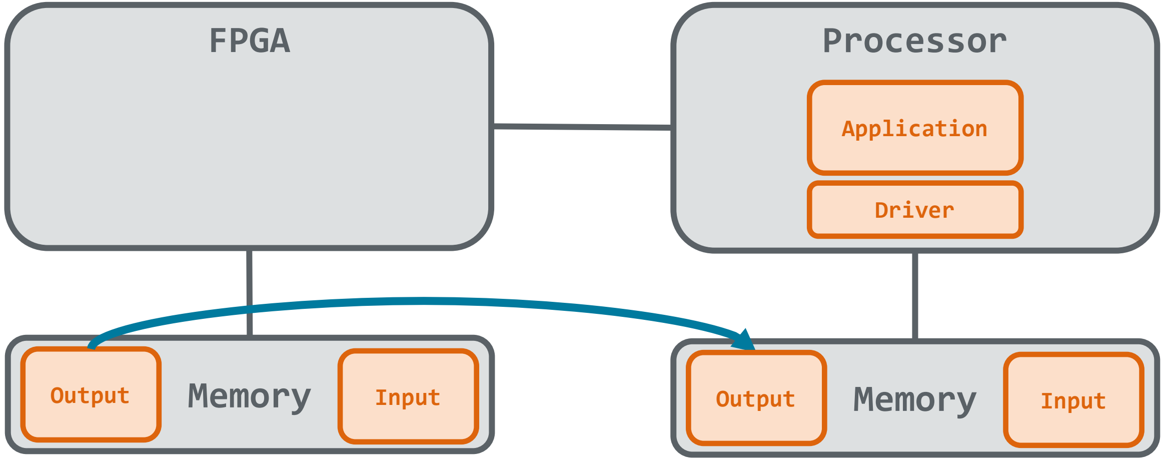


1. Initiate
2. Copy
3. **Process**
4. Copy
5. Complete

FPGA Accelerators

Device Attached Accelerators:

- Accelerator acts as a device in host system
- Accelerator can only access local resources
  - Host must copy data via DMA

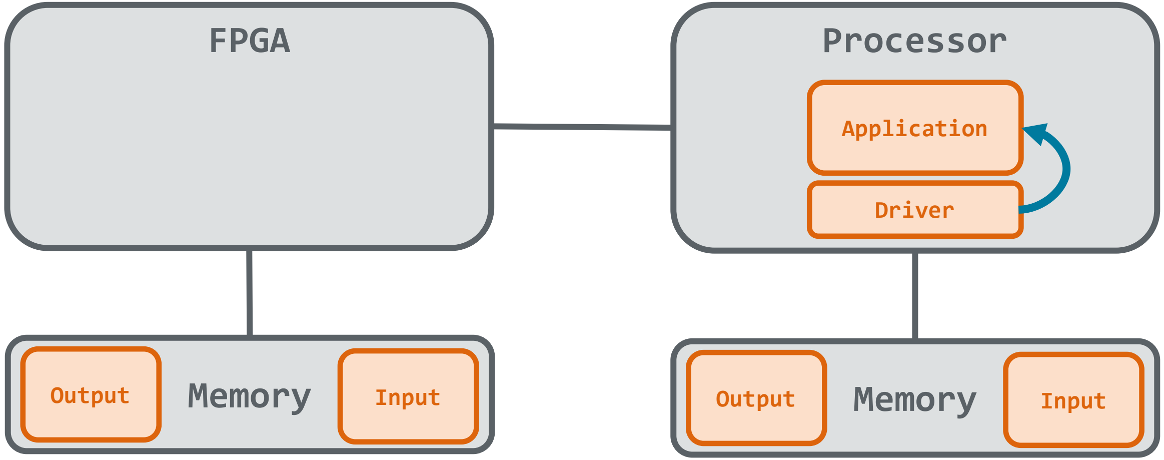


1. Initiate
2. Copy
3. Process
4. **Copy**
5. Complete

FPGA Accelerators

Device Attached Accelerators:

- Accelerator acts as a device in host system
- Accelerator can only access local resources
  - Host must copy data via DMA

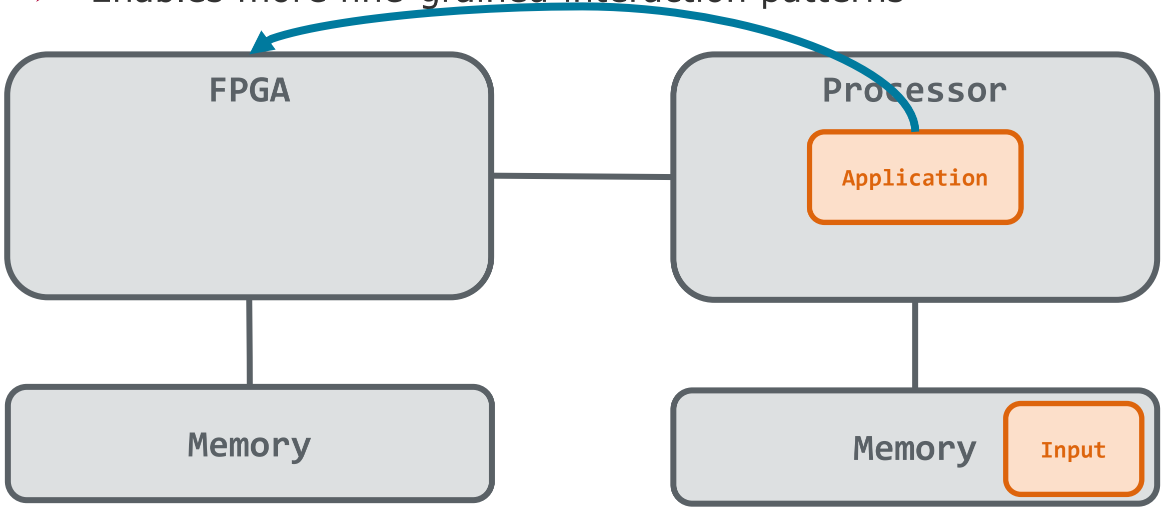


1. Initiate
2. Copy
3. Process
4. Copy
5. **Complete**

FPGA Accelerators

Coherently Attached Accelerators:

- Accelerator connected to the coherent memory interconnect on the host system
  - CAPI (OpenPOWER), CCIX (ARM), Gen-Z, CXL (Intel)
- Accelerator can autonomously access host memory
  - Enables more fine-grained interaction patterns

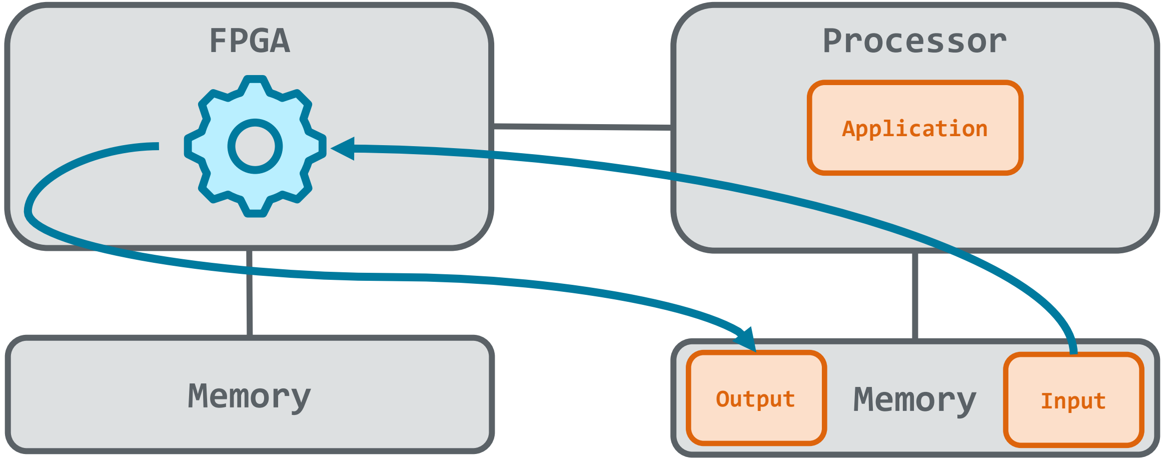


1. **Initiate**
2. Process
3. Complete

FPGA Accelerators

Coherently Attached Accelerators:

- Accelerator connected to the coherent memory interconnect on the host system
  - CAPI (OpenPOWER), CCIX (ARM), Gen-Z, CXL (Intel)
- Accelerator can autonomously access host memory
  - ➢ Enables more fine-grained interaction patterns

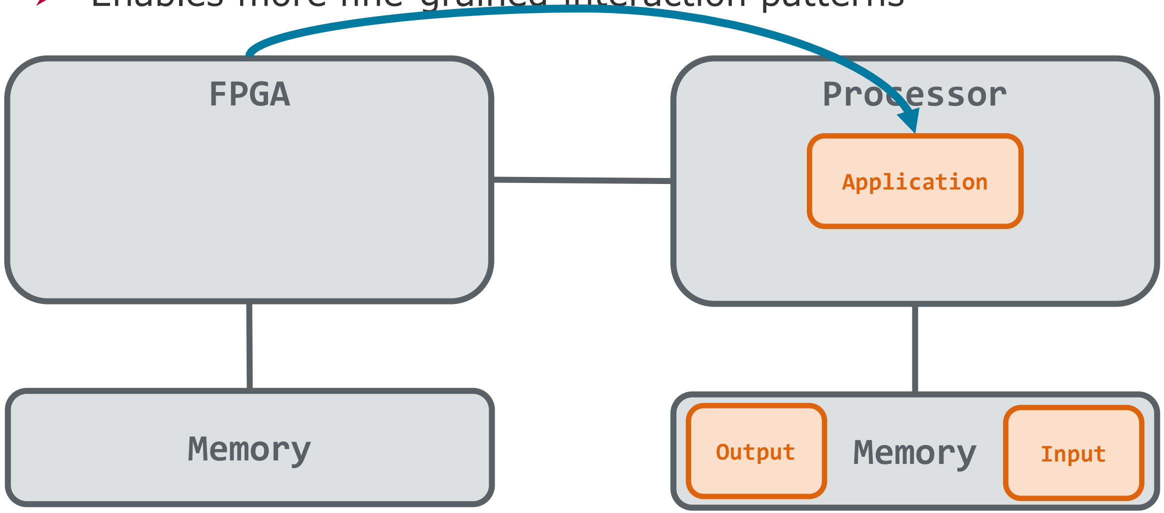


1. Initiate
2. **Process**
3. Complete

FPGA Accelerators

Coherently Attached Accelerators:

- Accelerator connected to the coherent memory interconnect on the host system
  - CAPI (OpenPOWER), CCIX (ARM), Gen-Z, CXL (Intel)
- Accelerator can autonomously access host memory
  - ➢ Enables more fine-grained interaction patterns



1. Initiate
2. Process
3. **Complete**

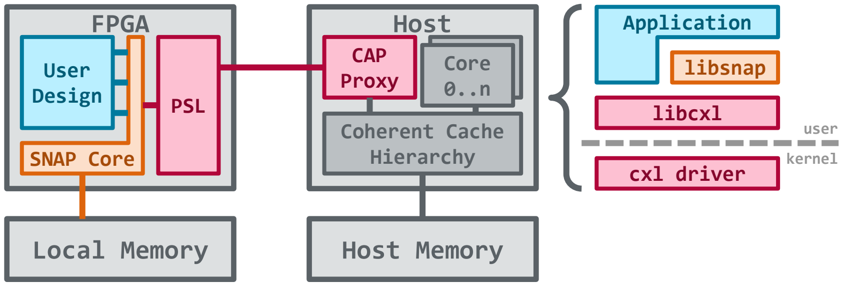
CAPI SNAP Framework

CAPI Interaction Scheme:

- Accelerator is **attached to a host process**
- Accelerator can access **virtual memory** space of host process
- Host process can access **control registers** exposed by the accelerator

SNAP Framework:

- Wraps low-level CAPI interface and local resources into a **homogeneous environment**

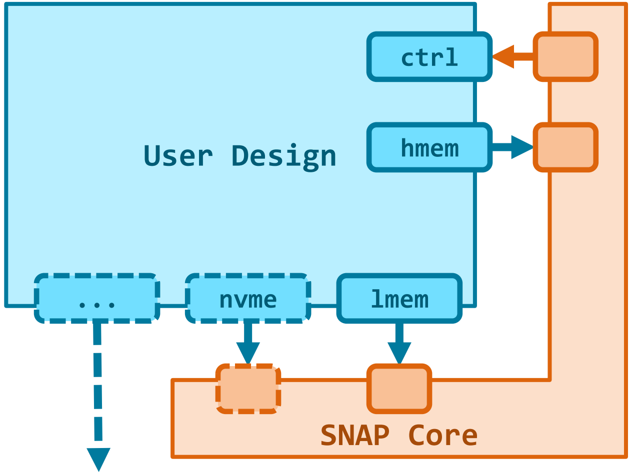


CAPI SNAP Framework

User Design Environment:

Consists of multiple random-access interfaces, each to a separate address space.

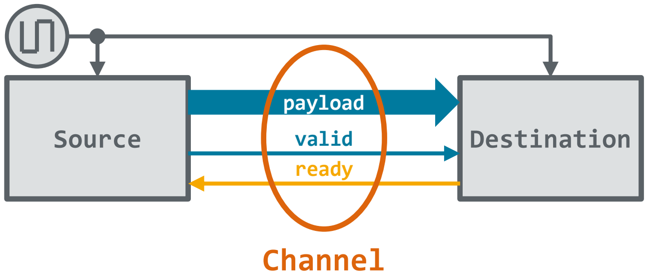
- Host Memory Interface, controlled by user design (master)
- Local Memory Interface, controlled by user design (master)
- Control Register Interface, controlled by host (slave)
  - Host writes configuration
  - Host reads status
  - Host can initiate user design activity by setting bits in specific control registers
- Optionally, SNAP can implement an NVMe controller to access non-volatile local storage
- Further card peripherals can be accessed via custom controllers

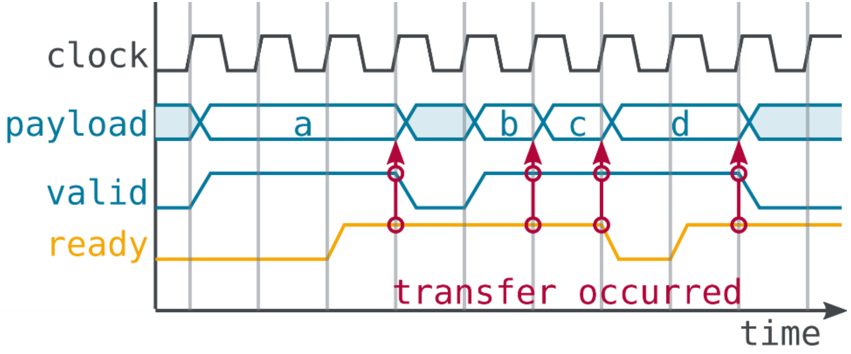


Excursion
AMBA Protocol Family

- The Advanced Microcontroller Bus Architecture (AMBA) was originally defined for ARM SoC designs → now widely adopted in FPGA designs

- **Channels** are a basic construct, used throughout the protocol family
  - **Payload** signals are transferred from a source to a destination
  - **Valid handshake** signal indicates that source presents new payload data
  - **Ready handshake** signal indicates that destination accepts transfer

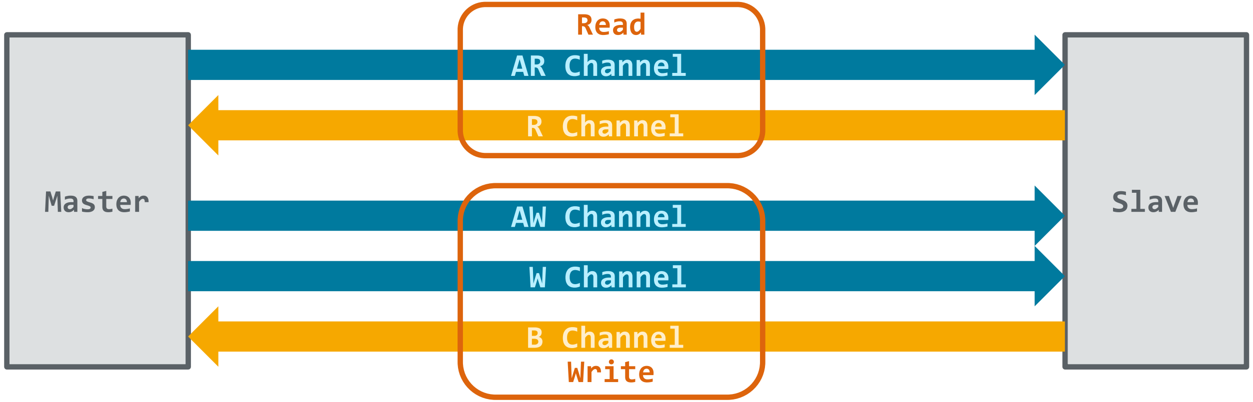




Excursion
AMBA Protocol Family

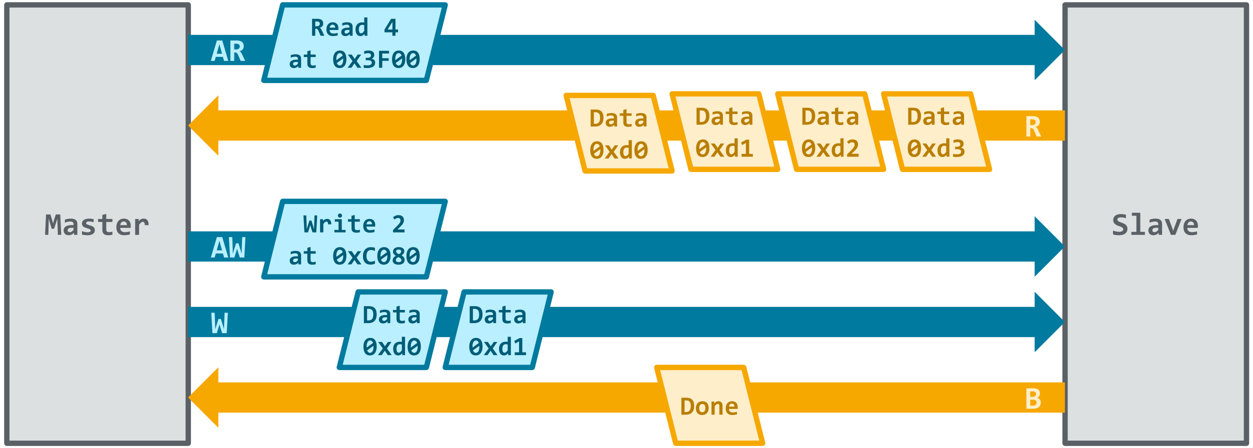
- The **Advanced Extensible Interface Stream** (AXI Stream) protocol uses a single AMBA channel to transmit sequential data streams from a master to a slave

- The **Advanced Extensible Interface** (AXI) protocol requires five AMBA channels to give a master random access to a slave address space



Excursion
AMBA Protocol Family

- AXI supports **burst transactions**:
  single read or write request initiates multiple contiguous data transfers



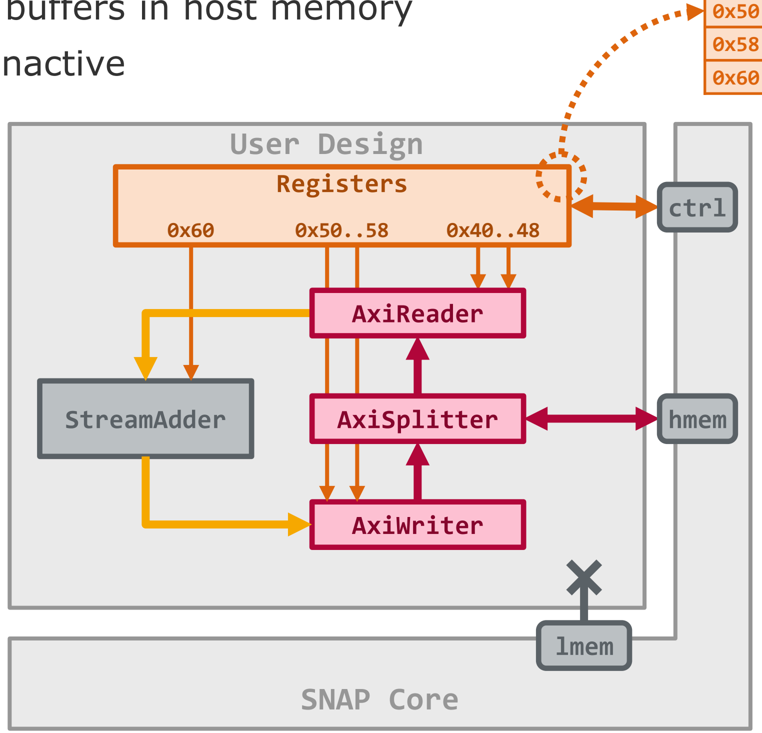
- **AXI Lite** is a simplified variant of the AXI protocol:
  - Same 5-channel structure
  - No burst capability
  - Suitable for peripheral register interfaces

Accelerator Design Example
A Data Stream Adder

Example: Add a configurable offset to a stream of unsigned 32bit integers

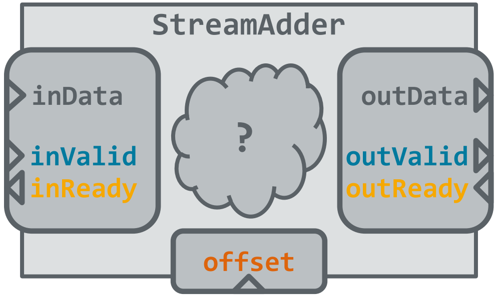
0x40..44	Read Address
0x48	Read Size (x64Byte)
0x50..54	Write Address
0x58	Write Size (x64Byte)
0x60	Offset Value

- Data stream is read from and written to buffers in host memory
  - `hmem` interface is used, `lmem` remains inactive
- Conversion between AXI and AXI Stream through `AxiReader` and `AxiWriter` modules
  - `AxiSplitter` separates read and write channels for both modules
- Actual implementation resides in `StreamAdder` module
- Control interface to host is realized in `Registers` module
  - Configures offset value and stream buffer addresses



Accelerator Design Example
A Data Stream Adder

```vhdl
entity StreamAdder is
  port (
    pi_clk       : in  std_logic;
    pi_rst_n     : in  std_logic;

    pi_offset    : in  unsigned (31 downto 0);

    pi_inData    : in  unsigned (511 downto 0);
    pi_inValid   : in  std_logic;
    po_inReady   : out std_logic;

    po_outData   : out unsigned (511 downto 0);
    po_outValid  : out std_logic;
    pi_outReady  : in  std_logic);
end StreamAdder;
```

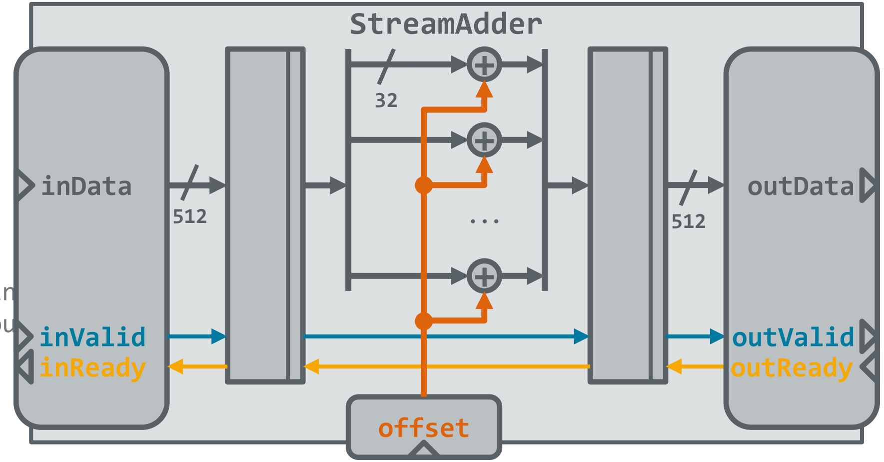


Accelerator Design Example
A Data Stream Adder

```
architecture StreamAdder of StreamAdder is
  signal s_data   : unsigned (511 downto 0);
  signal s_result : unsigned (511 downto 0);
  signal s_valid  : std_logic;
  signal s_ready  : std_logic;
begin
  i_inputStage : entity work.PipelineStage
    port map (pi_clk => pi_clk, pi_rst_n => pi_rst_n,
      pi_inData  => pi_inData,  pi_inValid  => pi_inValid,  po_inReady  => po_inReady,
      po_outData => s_data,     po_outValid => s_valid,     pi_outReady => s_ready);

  process(s_data)
  begin
    for v_idx in 0 to 15 loop
      s_result(v_idx*32+31 downto v_idx*32) <= s_data(v_idx*32+31 downto v_idx*32) + pi_offset;
    end loop;
  end process;

  i_outputStage : entity work.PipelineStage
    port map (pi_clk => pi_clk, pi_rst_n => pi_rst_n,
      pi_inData  => s_result,   pi_inValid  => s_valid,     po_inReady  => s_ready,
      po_outData => po_outData, po_outValid => po_outValid, pi_outReady => pi_outReady);
end StreamAdder;
```

Accelerator Design Example
A Data Stream Adder

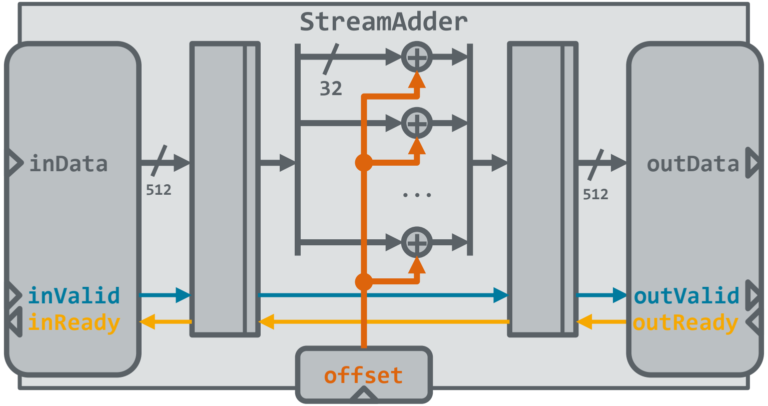
```
architecture StreamAdder of StreamAdder is
  signal s_data   : unsigned (511 downto 0);
  signal s_result : unsigned (511 downto 0);
  signal s_valid  : std_logic;
  signal s_ready  : std_logic;
begin
  i_inputStage : entity work.PipelineStage
    port map (pi_clk => pi_clk, pi_rst_n => pi_rst_n,
      pi_inData  => pi_inData,  pi_inValid  => pi_inValid,   po_in
      po_outData => s_data,     po_outValid => s_valid,      pi_ou

  process(s_data)
  begin
    for v_idx in 0 to 15 loop
      s_result(v_idx*32+31 downto v_idx*32) <= s_data(v_idx*32+31 downto v_idx*32) + pi_offset;
    end loop;
  end process;

  i_outputStage : entity work.PipelineStage
    port map (pi_clk => pi_clk, pi_rst_n => pi_rst_n,
      pi_inData  => s_result,   pi_inValid  => s_valid,      po_inReady  => s_ready,
      po_outData => po_outData, po_outValid => po_outValid, pi_outReady => pi_outReady);
end StreamAdder;
```



Accelerator Design Example
A Data Stream Adder

HLS Implementation

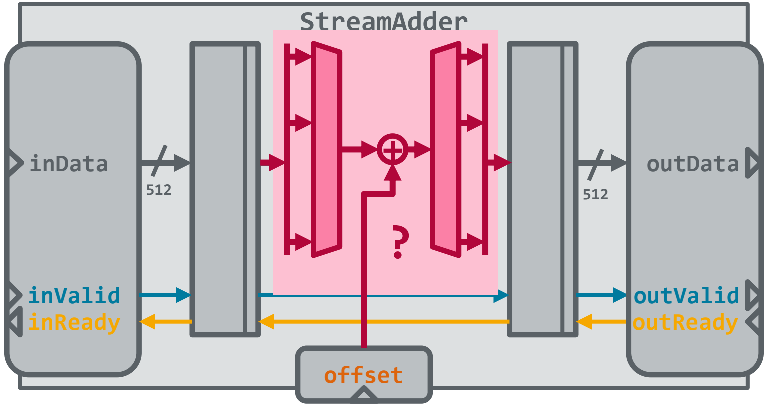
```
void StreamAdder(stream &in, stream &out, uint32_t offset) {
#pragma HLS INTERFACE axis      port=in     name=axis_input
#pragma HLS INTERFACE axis      port=out    name=axis_output
#pragma HLS INTERFACE s_axilite port=offset bundle=control offset=0x60
#pragma HLS INTERFACE s_axilite port=return bundle=control

  stream_element element;
  do {
    element = in.read();

    for (int i = 0; i < 16; ++i) {
      auto current = element.data(i * 32 + 31, i * 32);
      element.data(i * 32 + 31, i * 32) = current + offset;
    }

    out.write(element);
  } while (!element.last);
}
```



Accelerator Design Example
A Data Stream Adder

HLS Implementation

```
void StreamAdder(stream &in, stream &out, uint32_t offset) {
#pragma HLS INTERFACE axis      port=in     name=axis_input
#pragma HLS INTERFACE axis      port=out    name=axis_output
#pragma HLS INTERFACE s_axilite port=offset bundle=control offset=0x60
#pragma HLS INTERFACE s_axilite port=return bundle=control

  stream_element element;
  do {
    element = in.read();

    for (int i = 0; i < 16; ++i) {
      auto current = element.data(i * 32 + 31, i * 32);
      element.data(i * 32 + 31, i * 32) = current + offset;
    }

    out.write(element);
  } while (!element.last);
}
```



Accelerator Design Example
Takeaways

- **AXI Streams are convenient and efficient to decompose a design**
- **Top-level descriptions of stream-based designs share a similar structure**
- **Host software interacts with the accelerator through low-level registers**

Metal FS

Metal FS is an FPGA accelerator framework developed at the OSM group.

Concepts:

- **Operators** consume, produce or transform a data stream
- **Crossbar Switch** defines operator execution order at runtime

- **AXI Streams are convenient and efficient to decompose a design**
  - Metal FS is built around data streams
- **Top-level descriptions of stream-based designs share a similar structure**
  - Metal FS is an FPGA overlay, providing common facilities by default
- **Host software interacts with the accelerator through low-level registers**
  - Metal FS maps the FPGA accelerator to a userspace filesystem

```
$ cat ~/test.bin | /fpga/op/stream_add --offset=108 > ~/out1.bin
```

And now for a break and another bowl of Bancha.

*or beverage of your choice

Parallel Programming and Heterogeneous Computing

FPGA Accelerators - Metal FS Hands-On

Max Plauth, Sven Köhler, Felix Eberhardt, *Lukas Wenzel* and Andreas Polze
Operating Systems and Middleware Group

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Getting started with Metal FS

The Metal FS project lives on GitHub:

`https://github.com/osmhpi/metalfs`

Watch the intro video first (link on `https://osm.hpi.de/parProg`)!

We are going to follow a modified version of this tutorial:

`https://metalfs.github.io/tutorial.html`

To follow along, you will need a platform with Docker and docker-compose.

Getting started with Metal FS

Build a Metal FS operator to filter unsigned 32-bit values.

```
00000004  00000008
0000000c  0000000f
00000001  00000007
0000000d  00000005
00000000  00000003
0000000a  00000002
00000006  0000000e
00000009  0000000b
```

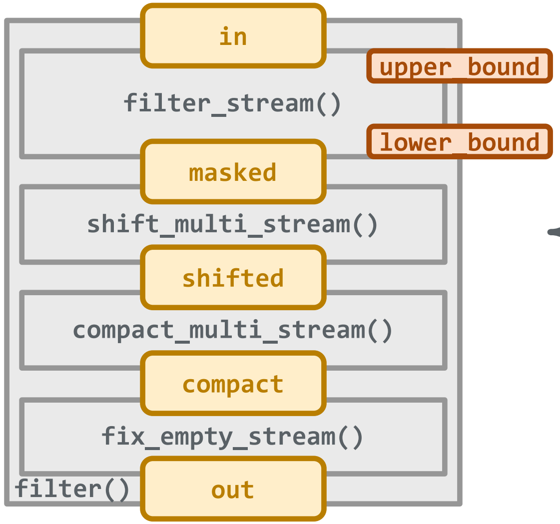
input.bin

```
$ cat input.bin
  | filter -l 4 -u 12
  > output.bin
```

```
00000004  00000008
0000000c  00000007
00000005  0000000a
00000006  00000009
0000000b
```

output.bin

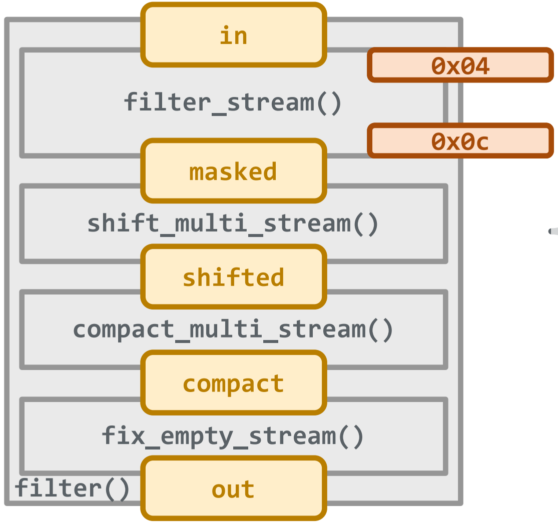
Getting started with Metal FS




```
void filter(mtl_stream &in, mtl_stream &out,
            Value lower_bound, Value upper_bound) {
#pragma HLS INTERFACE axis port=in name=axis_input
#pragma HLS INTERFACE axis port=out name=axis_output
#pragma HLS INTERFACE s_axilite port=lower_bound bundle=control offset=0x100
#pragma HLS INTERFACE s_axilite port=upper_bound bundle=control offset=0x110
#pragma HLS INTERFACE s_axilite port=return bundle=control
  MaskedStream masked;
  ShiftedStream shifted;
  mtl_stream compact;

#pragma HLS DATAFLOW

  filter_stream(in, masked, lower_bound, upper_bound);

#if VALUE_COUNT > 1
  shift_multi_stream(masked, shifted);
  compact_multi_stream(shifted, compact);
#else
  compact_single_stream(masked, compact);
#endif

  fix_empty_stream(compact, out);
}
```

Getting started with Metal FS

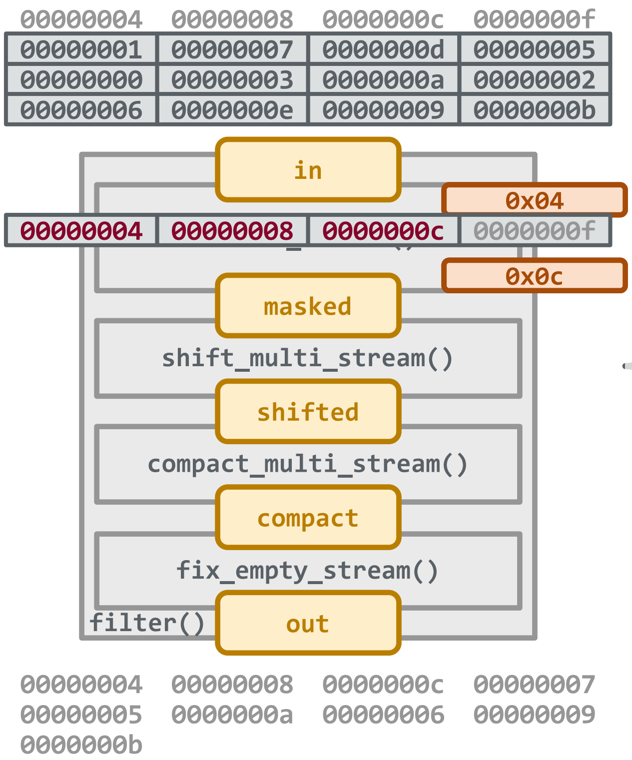
00000004	00000008	0000000c	0000000f
00000001	00000007	0000000d	00000005
00000000	00000003	0000000a	00000002
00000006	0000000e	00000009	0000000b



00000004 00000008 0000000c 00000007
00000005 0000000a 00000006 00000009
0000000b

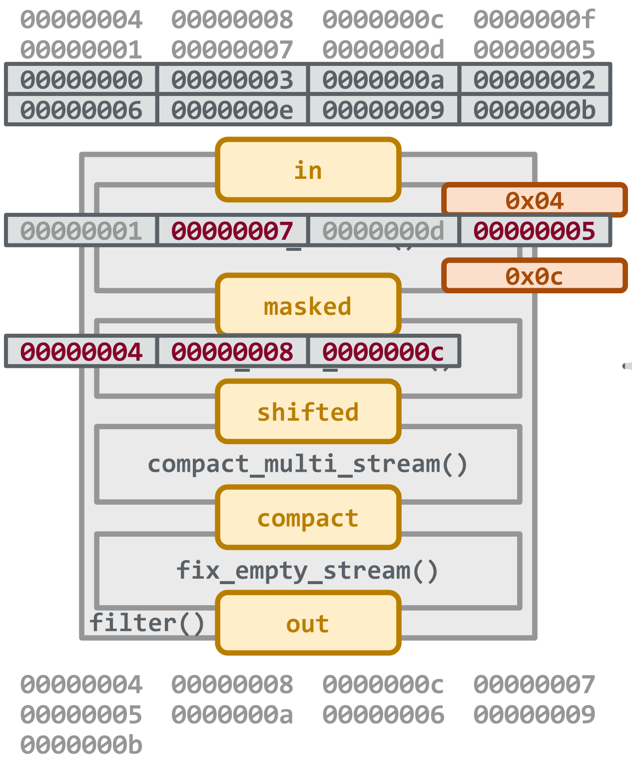
```
void filter(mtl_stream &in, mtl_stream &out,
            Value lower_bound, Value upper_bound) {
#pragma HLS INTERFACE axis port=in name=axis_input
#pragma HLS INTERFACE axis port=out name=axis_output
#pragma HLS INTERFACE s_axilite port=lower_bound bundle=control offset=0x100
#pragma HLS INTERFACE s_axilite port=upper_bound bundle=control offset=0x110
#pragma HLS INTERFACE s_axilite port=return bundle=control
  MaskedStream masked;
  ShiftedStream shifted;
  mtl_stream compact;

#pragma HLS DATAFLOW

  filter_stream(in, masked, lower_bound, upper_bound);

#if VALUE_COUNT > 1
  shift_multi_stream(masked, shifted);
  compact_multi_stream(shifted, compact);
#else
  compact_single_stream(masked, compact);
#endif

  fix_empty_stream(compact, out);
}
```

Getting started with Metal FS



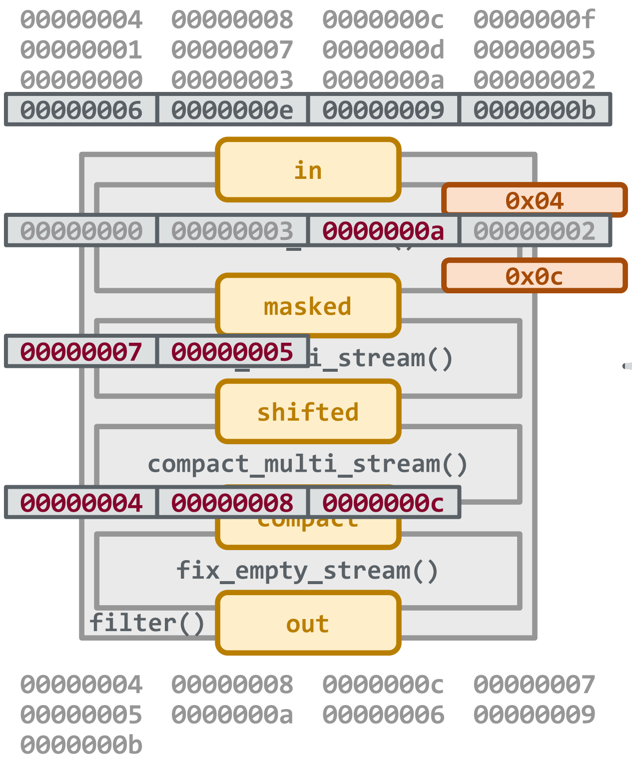

```
void filter(mtl_stream &in, mtl_stream &out,
            Value lower_bound, Value upper_bound) {
#pragma HLS INTERFACE axis port=in name=axis_input
#pragma HLS INTERFACE axis port=out name=axis_output
#pragma HLS INTERFACE s_axilite port=lower_bound bundle=control offset=0x100
#pragma HLS INTERFACE s_axilite port=upper_bound bundle=control offset=0x110
#pragma HLS INTERFACE s_axilite port=return bundle=control
  MaskedStream masked;
  ShiftedStream shifted;
  mtl_stream compact;

#pragma HLS DATAFLOW

  filter_stream(in, masked, lower_bound, upper_bound);

#if VALUE_COUNT > 1
  shift_multi_stream(masked, shifted);
  compact_multi_stream(shifted, compact);
#else
  compact_single_stream(masked, compact);
#endif

  fix_empty_stream(compact, out);
}
```

Getting started with Metal FS



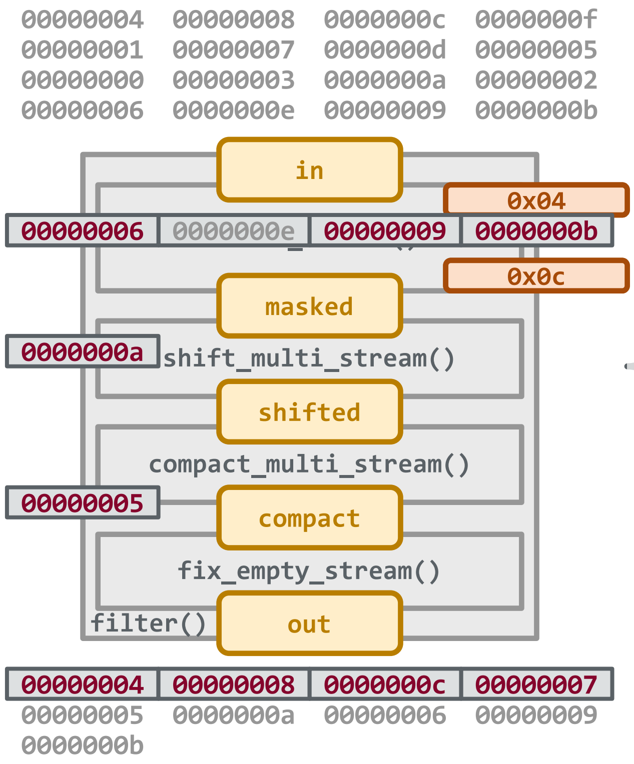

```
void filter(mtl_stream &in, mtl_stream &out,
            Value lower_bound, Value upper_bound) {
#pragma HLS INTERFACE axis port=in name=axis_input
#pragma HLS INTERFACE axis port=out name=axis_output
#pragma HLS INTERFACE s_axilite port=lower_bound bundle=control offset=0x100
#pragma HLS INTERFACE s_axilite port=upper_bound bundle=control offset=0x110
#pragma HLS INTERFACE s_axilite port=return bundle=control
  MaskedStream masked;
  ShiftedStream shifted;
  mtl_stream compact;

#pragma HLS DATAFLOW

  filter_stream(in, masked, lower_bound, upper_bound);

#if VALUE_COUNT > 1
  shift_multi_stream(masked, shifted);
  compact_multi_stream(shifted, compact);
#else
  compact_single_stream(masked, compact);
#endif

  fix_empty_stream(compact, out);
}
```

Getting started with Metal FS



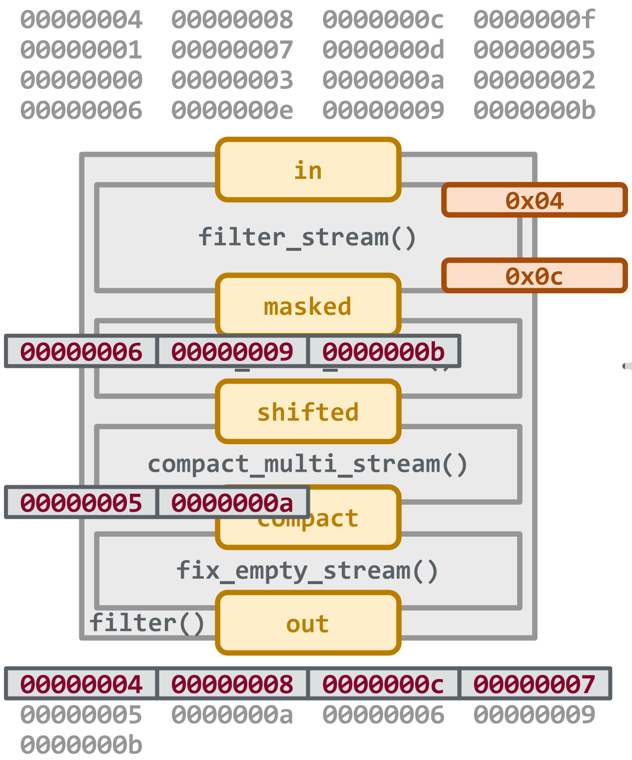

```
void filter(mtl_stream &in, mtl_stream &out,
            Value lower_bound, Value upper_bound) {
#pragma HLS INTERFACE axis port=in name=axis_input
#pragma HLS INTERFACE axis port=out name=axis_output
#pragma HLS INTERFACE s_axilite port=lower_bound bundle=control offset=0x100
#pragma HLS INTERFACE s_axilite port=upper_bound bundle=control offset=0x110
#pragma HLS INTERFACE s_axilite port=return bundle=control
  MaskedStream masked;
  ShiftedStream shifted;
  mtl_stream compact;

#pragma HLS DATAFLOW

  filter_stream(in, masked, lower_bound, upper_bound);

#if VALUE_COUNT > 1
  shift_multi_stream(masked, shifted);
  compact_multi_stream(shifted, compact);
#else
  compact_single_stream(masked, compact);
#endif

  fix_empty_stream(compact, out);
}
```

Getting started with Metal FS



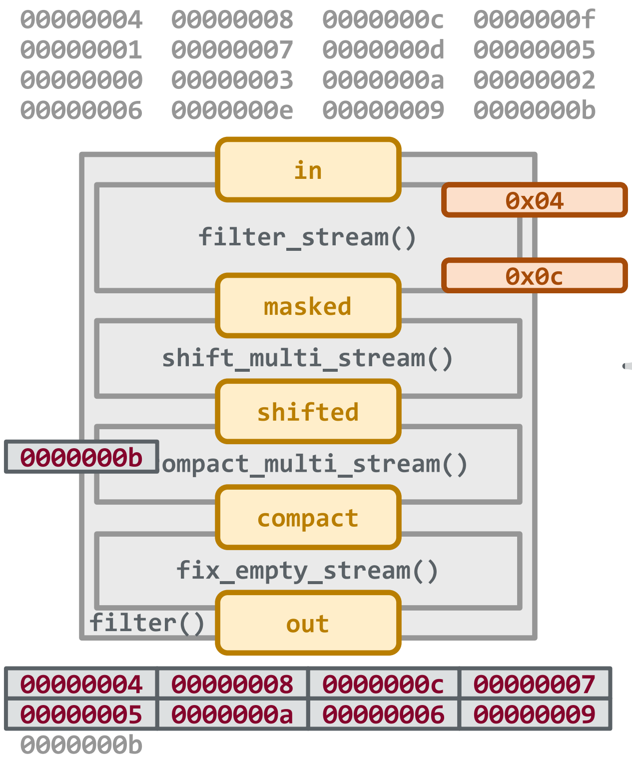

```
void filter(mtl_stream &in, mtl_stream &out,
            Value lower_bound, Value upper_bound) {
#pragma HLS INTERFACE axis port=in name=axis_input
#pragma HLS INTERFACE axis port=out name=axis_output
#pragma HLS INTERFACE s_axilite port=lower_bound bundle=control offset=0x100
#pragma HLS INTERFACE s_axilite port=upper_bound bundle=control offset=0x110
#pragma HLS INTERFACE s_axilite port=return bundle=control
  MaskedStream masked;
  ShiftedStream shifted;
  mtl_stream compact;

#pragma HLS DATAFLOW

  filter_stream(in, masked, lower_bound, upper_bound);

#if VALUE_COUNT > 1
  shift_multi_stream(masked, shifted);
  compact_multi_stream(shifted, compact);
#else
  compact_single_stream(masked, compact);
#endif

  fix_empty_stream(compact, out);
}
```

Getting started with Metal FS




```
void filter(mtl_stream &in, mtl_stream &out,
            Value lower_bound, Value upper_bound) {
#pragma HLS INTERFACE axis port=in name=axis_input
#pragma HLS INTERFACE axis port=out name=axis_output
#pragma HLS INTERFACE s_axilite port=lower_bound bundle=control offset=0x100
#pragma HLS INTERFACE s_axilite port=upper_bound bundle=control offset=0x110
#pragma HLS INTERFACE s_axilite port=return bundle=control
  MaskedStream masked;
  ShiftedStream shifted;
  mtl_stream compact;

#pragma HLS DATAFLOW

  filter_stream(in, masked, lower_bound, upper_bound);

#if VALUE_COUNT > 1
  shift_multi_stream(masked, shifted);
  compact_multi_stream(shifted, compact);
#else
  compact_single_stream(masked, compact);
#endif

  fix_empty_stream(compact, out);
}
```

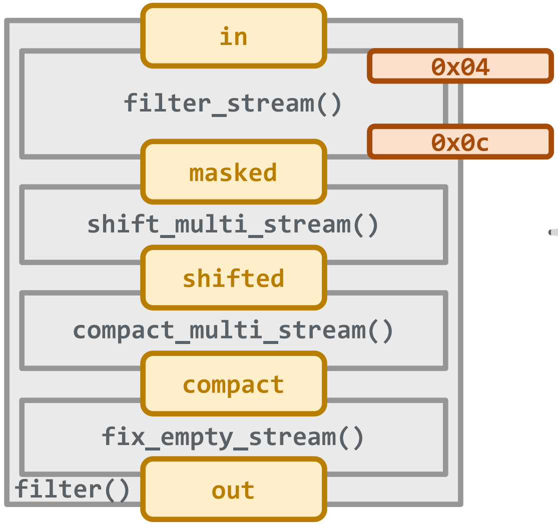
Getting started with Metal FS




```
void filter(mtl_stream &in, mtl_stream &out,
            Value lower_bound, Value upper_bound) {
#pragma HLS INTERFACE axis port=in name=axis_input
#pragma HLS INTERFACE axis port=out name=axis_output
#pragma HLS INTERFACE s_axilite port=lower_bound bundle=control offset=0x100
#pragma HLS INTERFACE s_axilite port=upper_bound bundle=control offset=0x110
#pragma HLS INTERFACE s_axilite port=return bundle=control
  MaskedStream masked;
  ShiftedStream shifted;
  mtl_stream compact;

#pragma HLS DATAFLOW

  filter_stream(in, masked, lower_bound, upper_bound);

#if VALUE_COUNT > 1
  shift_multi_stream(masked, shifted);
  compact_multi_stream(shifted, compact);
#else
  compact_single_stream(masked, compact);
#endif

  fix_empty_stream(compact, out);
}
```

Getting started with Metal FS

00000004	00000008	0000000c	0000000f
00000001	00000007	0000000d	00000005
00000000	00000003	0000000a	00000002
00000006	0000000e	00000009	0000000b



00000004	00000008	0000000c	00000007
00000005	0000000a	00000006	00000009
0000000b			

```
void filter(mtl_stream &in, mtl_stream &out,
            Value lower_bound, Value upper_bound) {
#pragma HLS INTERFACE axis port=in name=axis_input
#pragma HLS INTERFACE axis port=out name=axis_output
#pragma HLS INTERFACE s_axilite port=lower_bound bundle=control offset=0x100
#pragma HLS INTERFACE s_axilite port=upper_bound bundle=control offset=0x110
#pragma HLS INTERFACE s_axilite port=return bundle=control
  MaskedStream masked;
  ShiftedStream shifted;
  mtl_stream compact;

#pragma HLS DATAFLOW

  filter_stream(in, masked, lower_bound, upper_bound);

#if VALUE_COUNT > 1
  shift_multi_stream(masked, shifted);
  compact_multi_stream(shifted, compact);
#else
  compact_single_stream(masked, compact);
#endif

  fix_empty_stream(compact, out);
}
```

And now for a break and some cold brewed Bancha.

*or beverage of your choice