Garnet: The Next Generation CGRA Architecture

Priyanka Raina

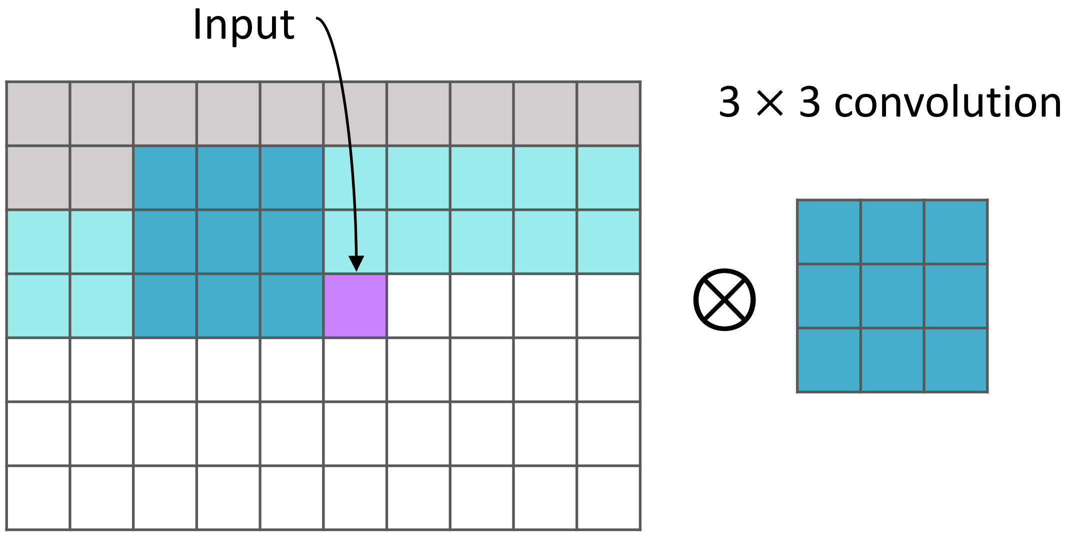
May 10, 2019

Challenges with Jade CGRA

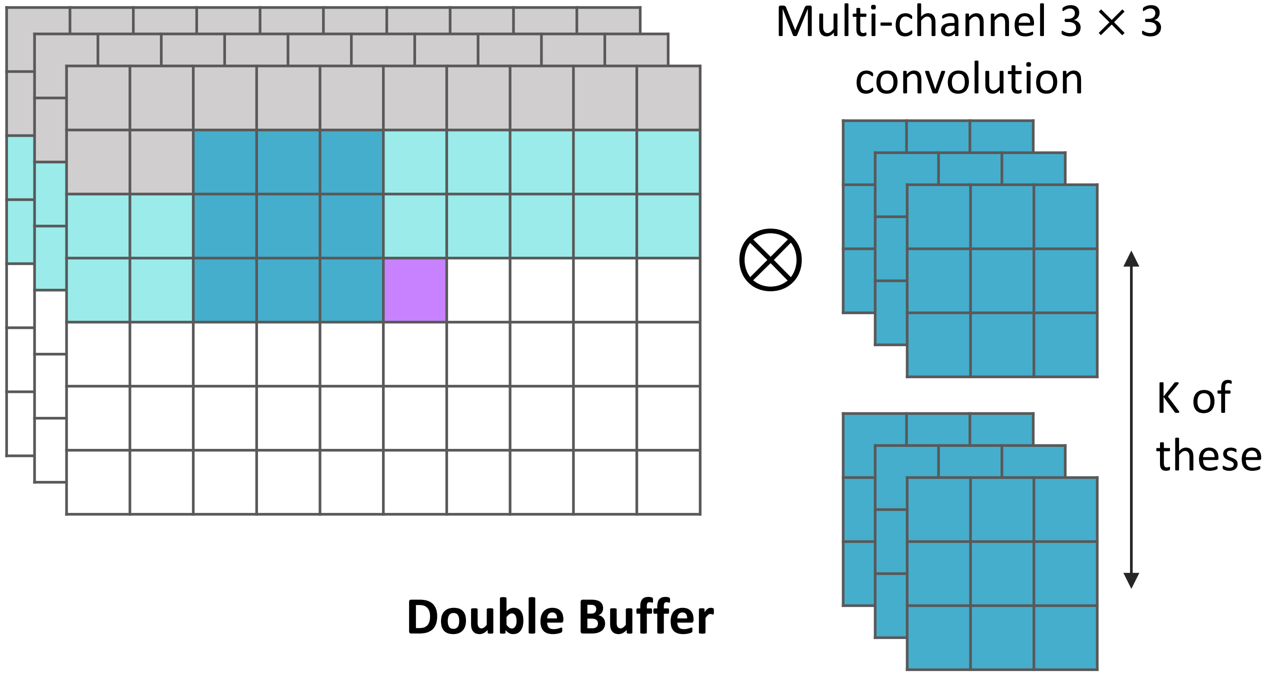
1. PE supports only simple integer operations
    - Porting applications that use floating point requires application expertise and manual effort

Challenges with Jade CGRA

2. Memory supports only line buffered pipelines
    - Most new applications even for imaging and vision use neural networks
    - Need a memory hierarchy with double buffers for energy-efficiency



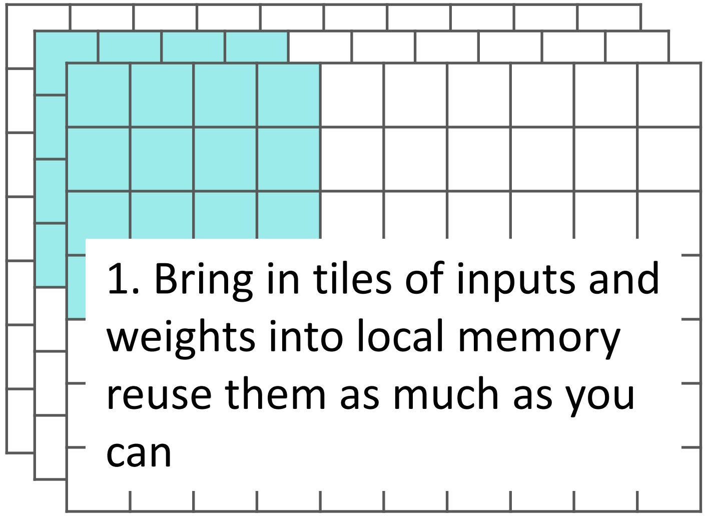
Line Buffer



Double Buffer

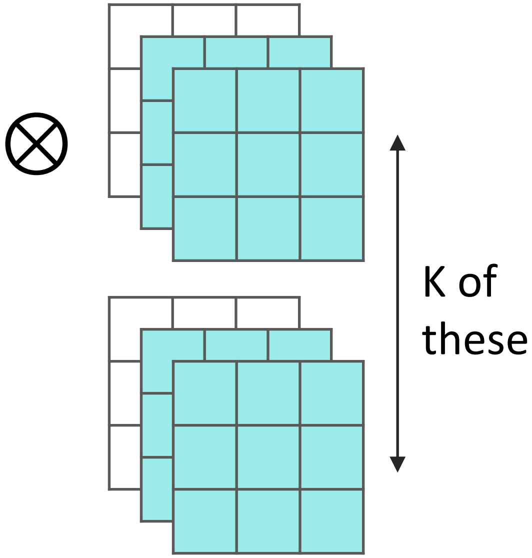
Challenges with Jade CGRA

2. Memory supports only line buffered pipelines
   - Most new applications even for imaging and vision use neural networks
   - Need a memory hierarchy with double buffers for energy-efficiency



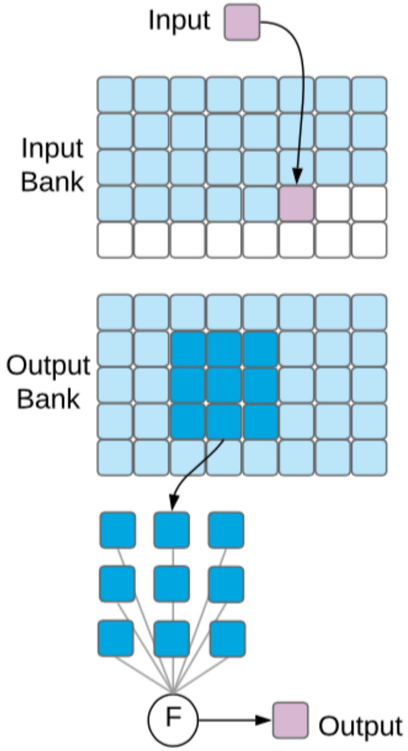
1. Bring in tiles of inputs and weights into local memory reuse them as much as you can

Multi-channel 3 × 3 convolution



2. Use a double buffer to overlap compute on current tile and fetching of next tile

3. Use a hierarchy of double buffers to maximize energy-efficiency



Challenges with Jade CGRA

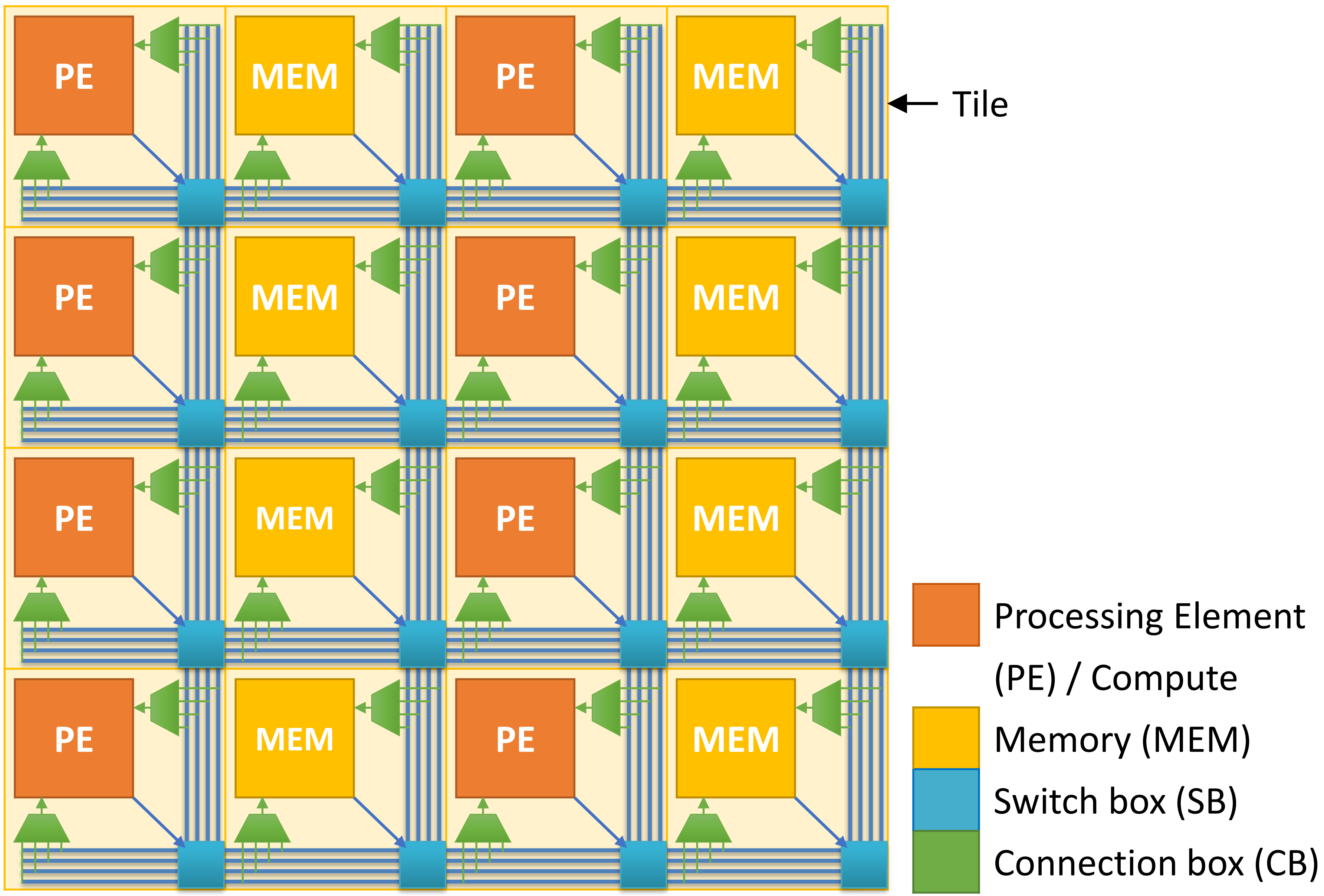
3. Configuration over JTAG is slow
    - Large applications with multiple kernels don’t fit on the CGRA – need fast reconfiguration

Key (application driven!) architectural changes in Garnet CGRA

1. Support for Bfloat16, and for executing complex operations like divide using multiple PEs
2. Addition of a global buffer to create a memory hierarchy for efficiently executing neural networks, and double buffer support in all memories
3. Fast reconfiguration support using global buffer and control processor
4. Addition of configurable power domains

CGRA

Coarse Grained Reconfigurable Array (CGRA)



🟧 Processing Element (PE) / Compute

🟨 Memory (MEM)

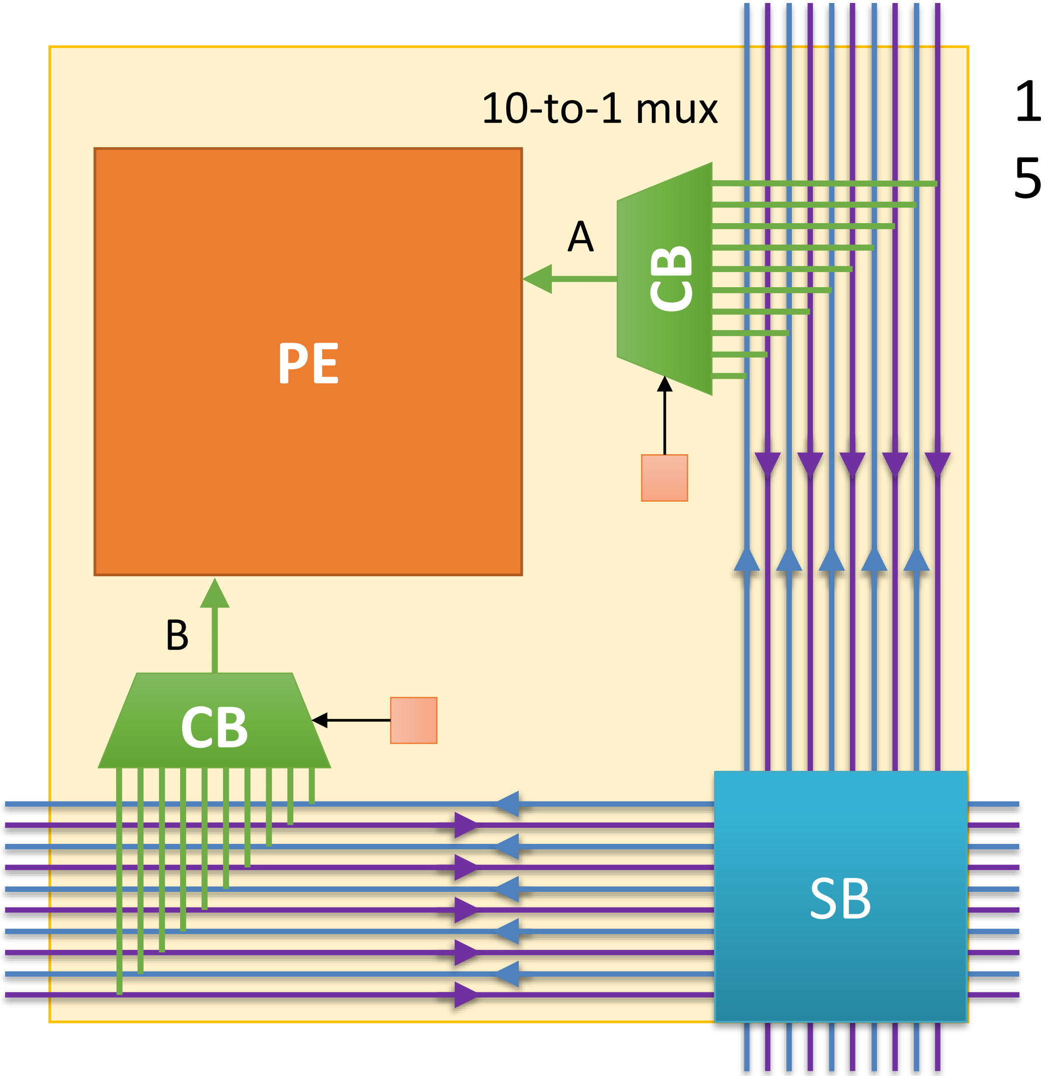
🟦 Switch box (SB)

🟩 Connection box (CB)

Interconnect

Keyi Zhang

Connection Box (CB) - Jade

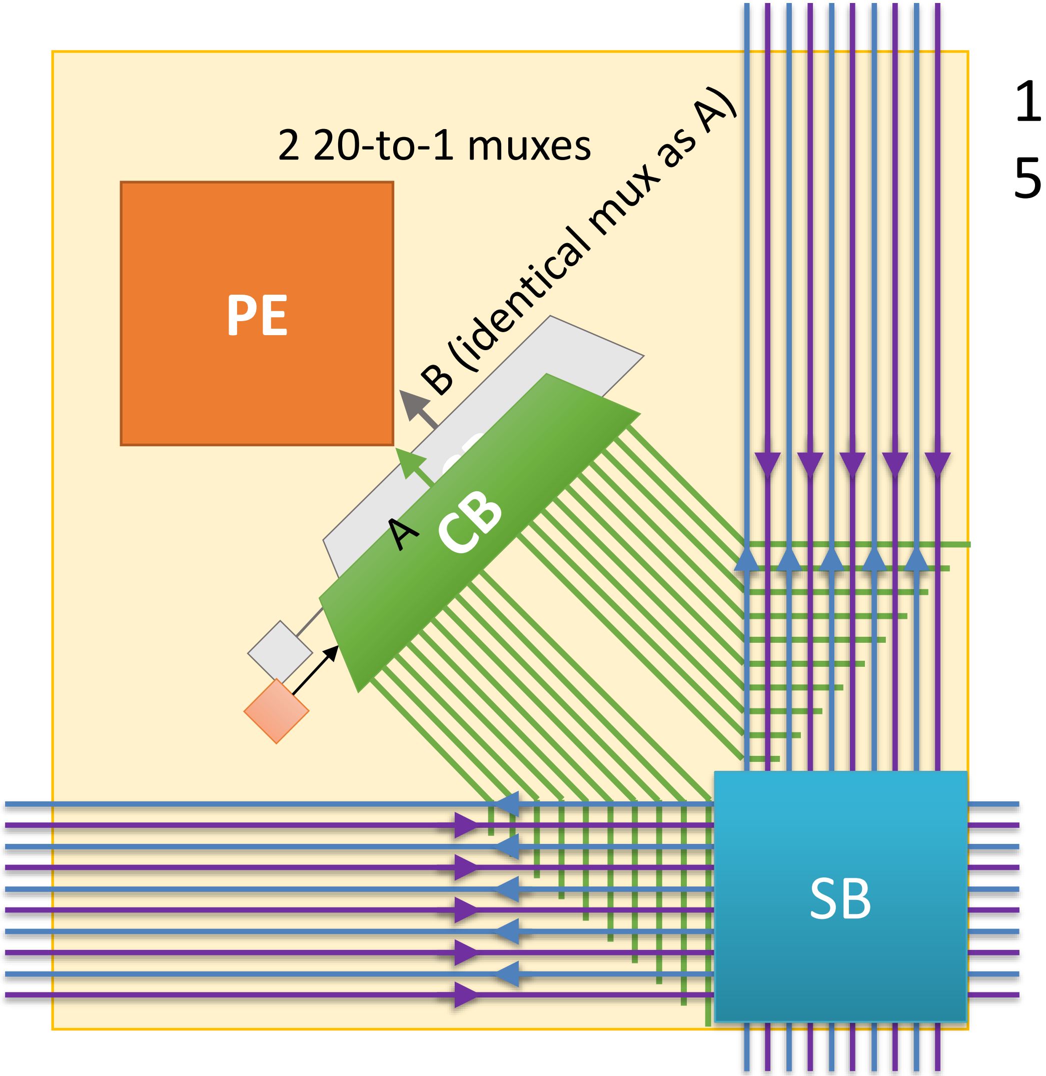


10 vertical 16 bit tracks
5 driven up, 5 driven down

There are similar 1 bit CBs for feeding 1 bit inputs from 10 horizontal and vertical 1 bit tracks

10 horizontal 16 bit tracks
5 driven left, 5 driven right

Connection Box (CB) - Garnet

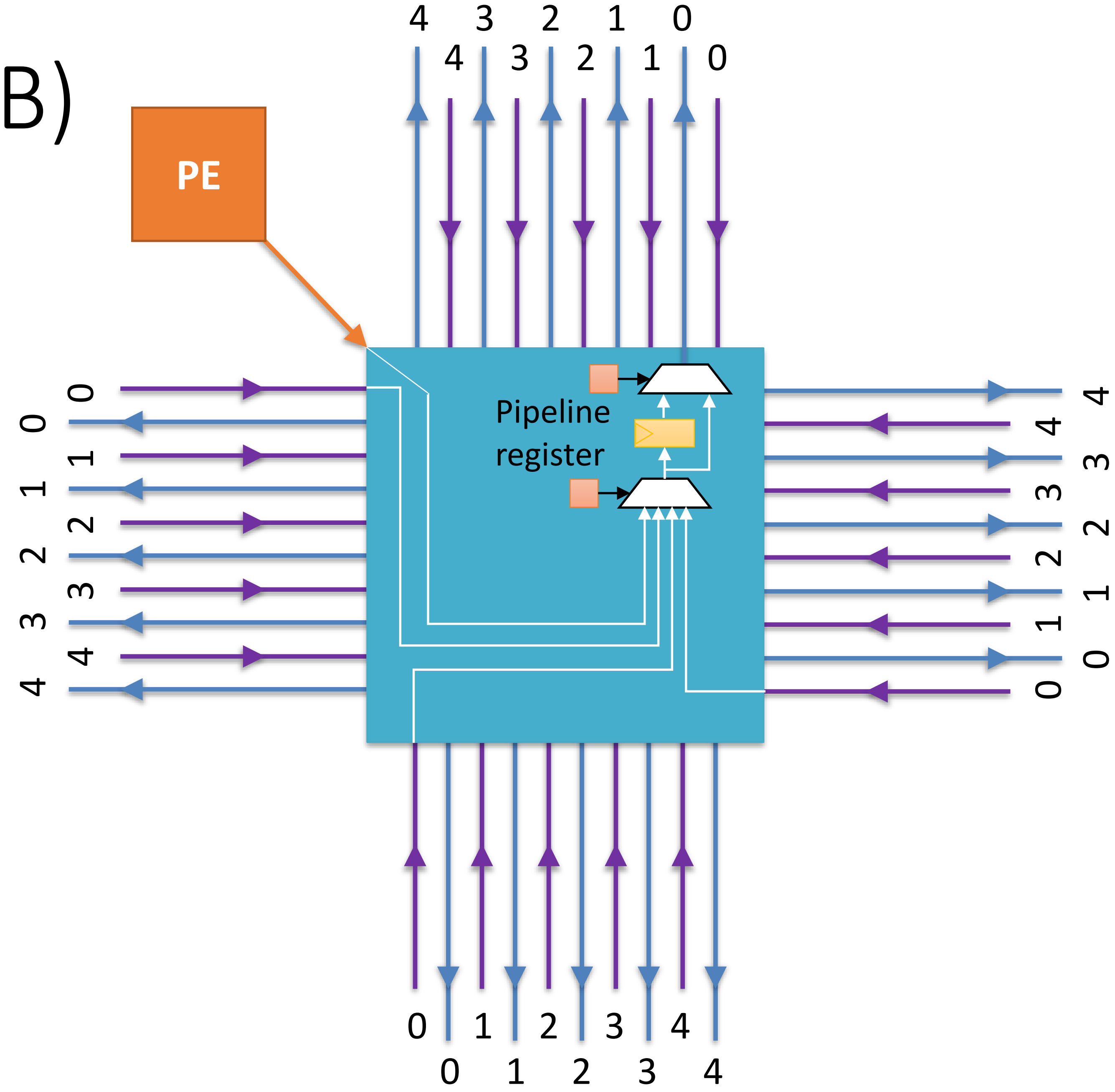


10 vertical 16 bit tracks
5 driven up, 5 driven down

10 horizontal 16 bit tracks
5 driven left, 5 driven right

There are similar 1 bit CBs for feeding 1 bit inputs from 10 horizontal and vertical 1 bit tracks

Switch Box (SB)



Switch Box (SB)

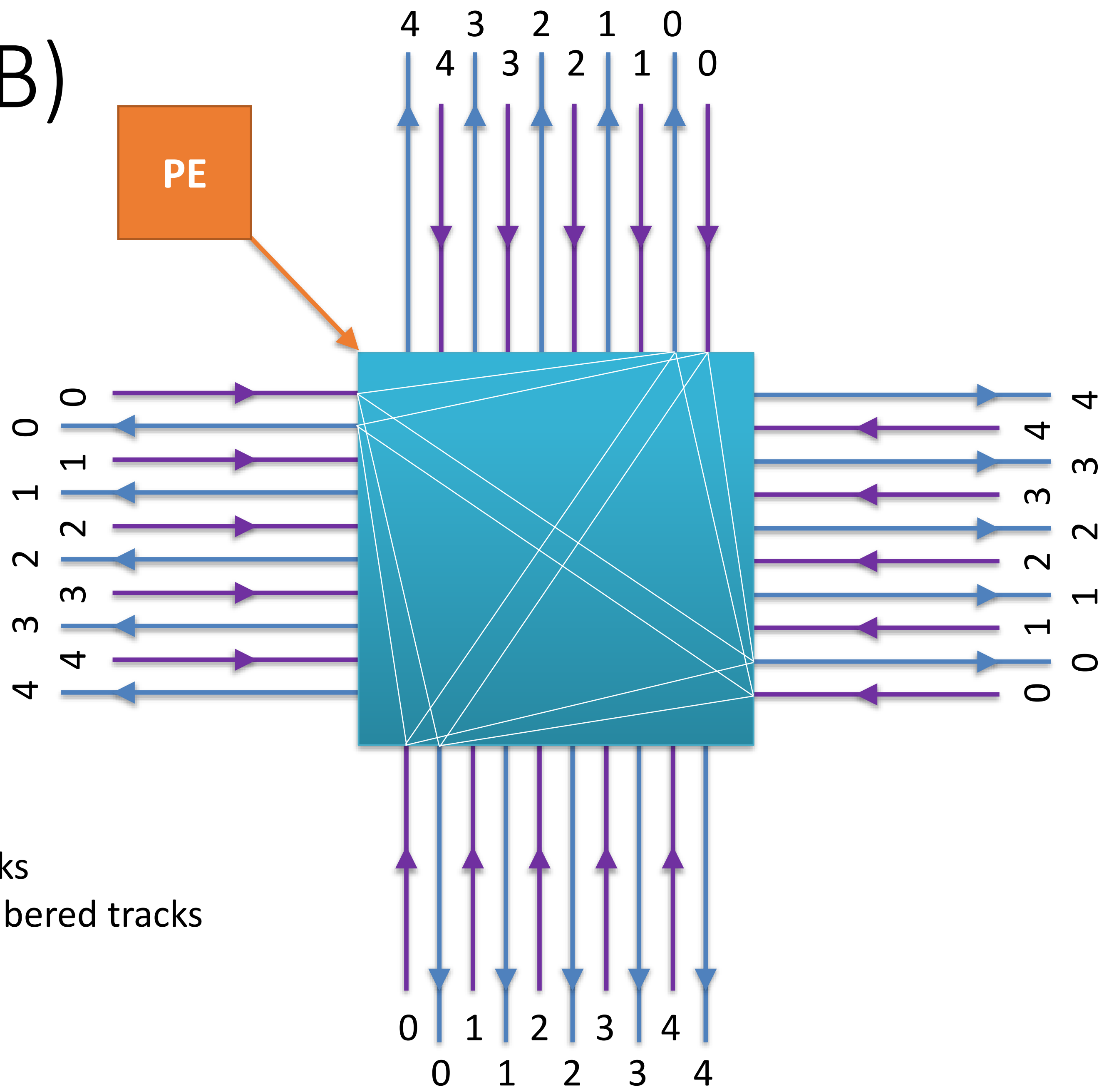
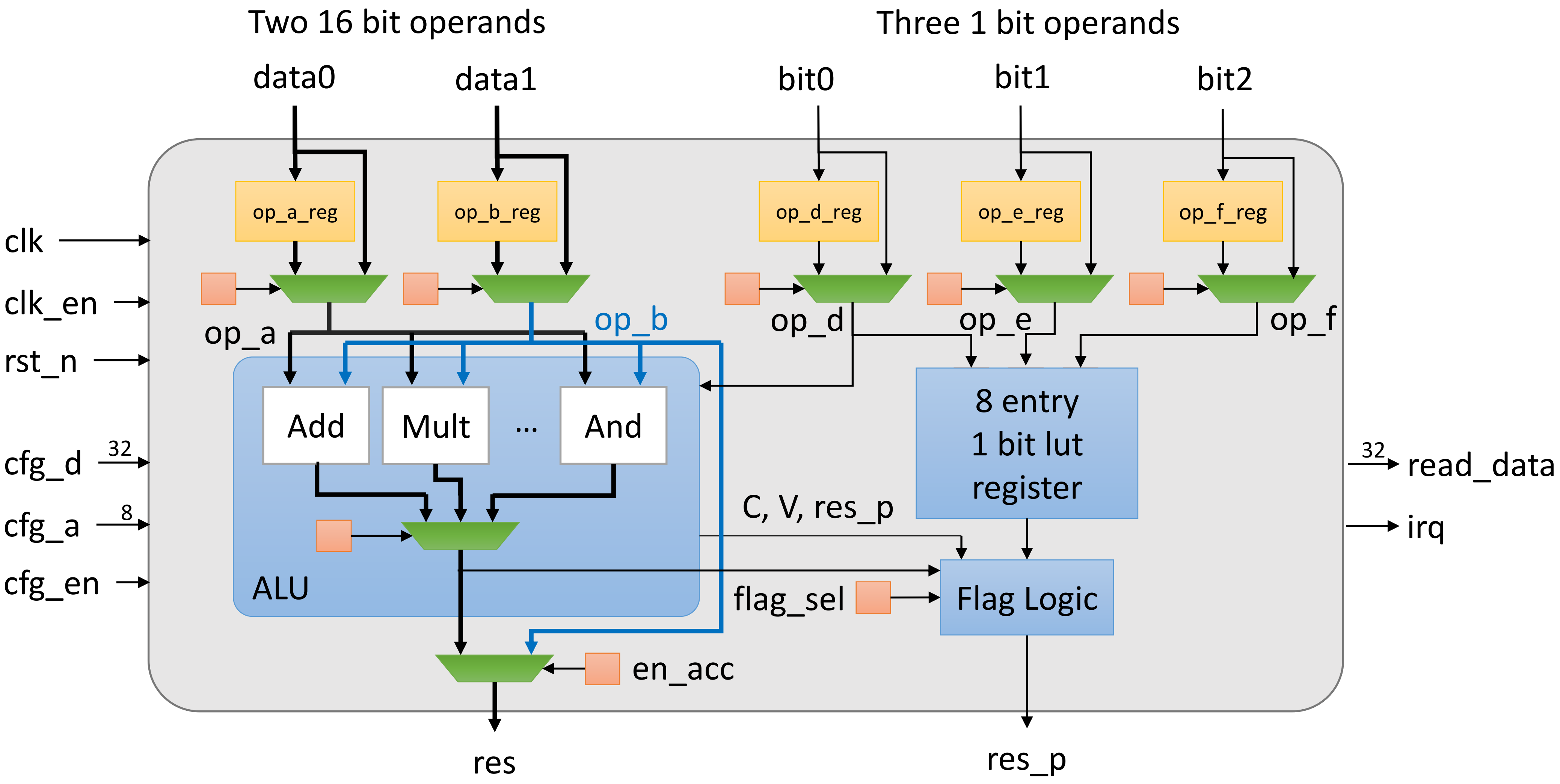


Figure shows connectivity for ‘0’ tracks
Similar connectivity for all other numbered tracks
PE output goes to all muxes

Processing Element

Nikhil Bhagdikar

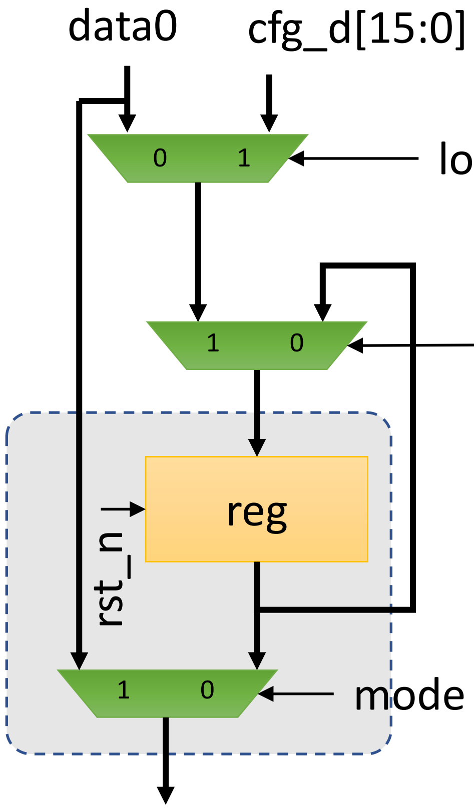
Processing Element (PE) in Diablo



Register Modes

Previous slide shows only this dotted portion

In reality, the register is more complex because of debug functionality



Or cfg_d[0] for 1 bit registers

This particular register’s address

load = cfg_en && (cfg_a == F0)

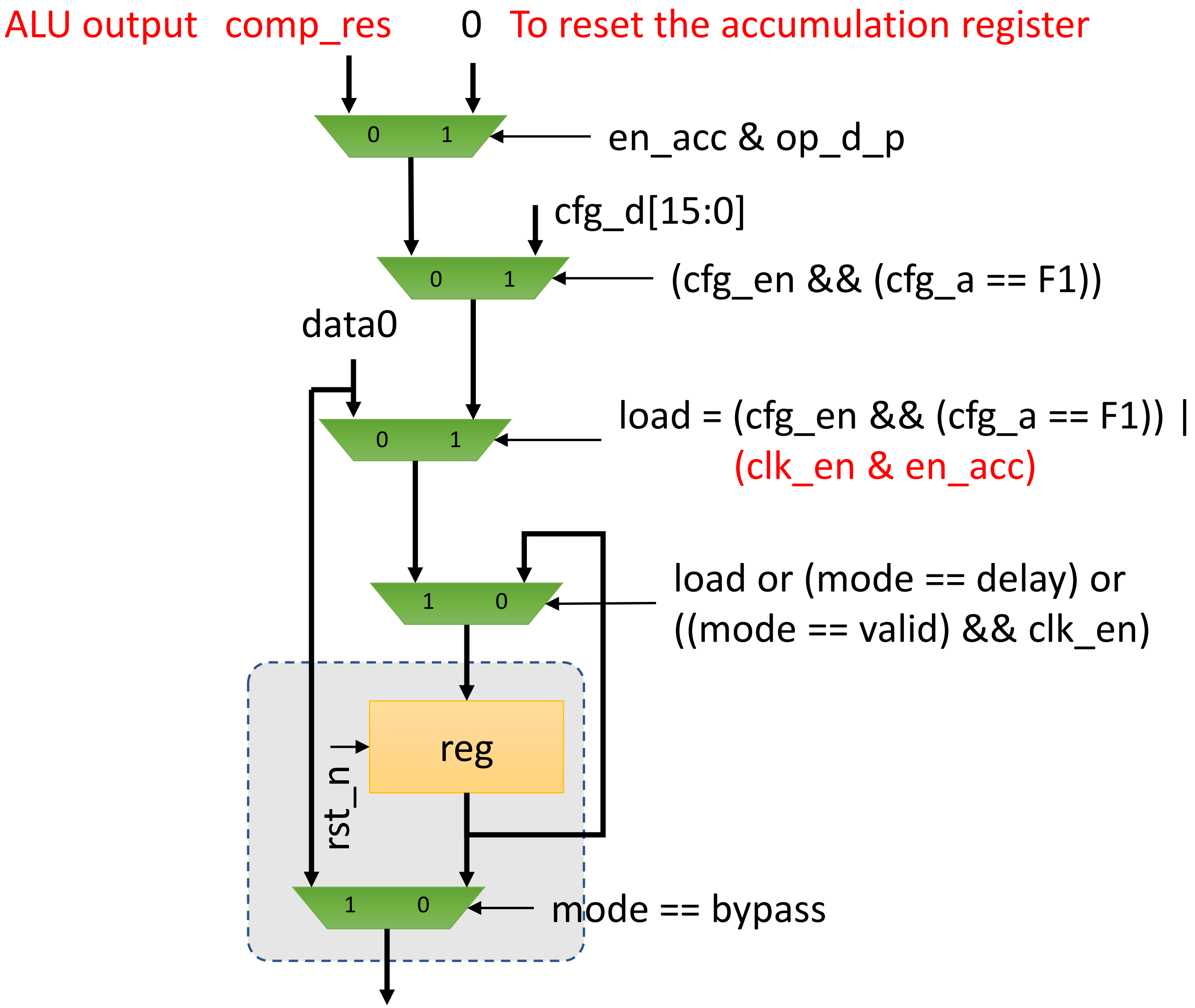
load or (mode == delay) or ((mode == valid) && clk_en)

In both modes, the data is registered, in delay mode it is unconditionally registered, in valid mode it is registered only when clk is enabled

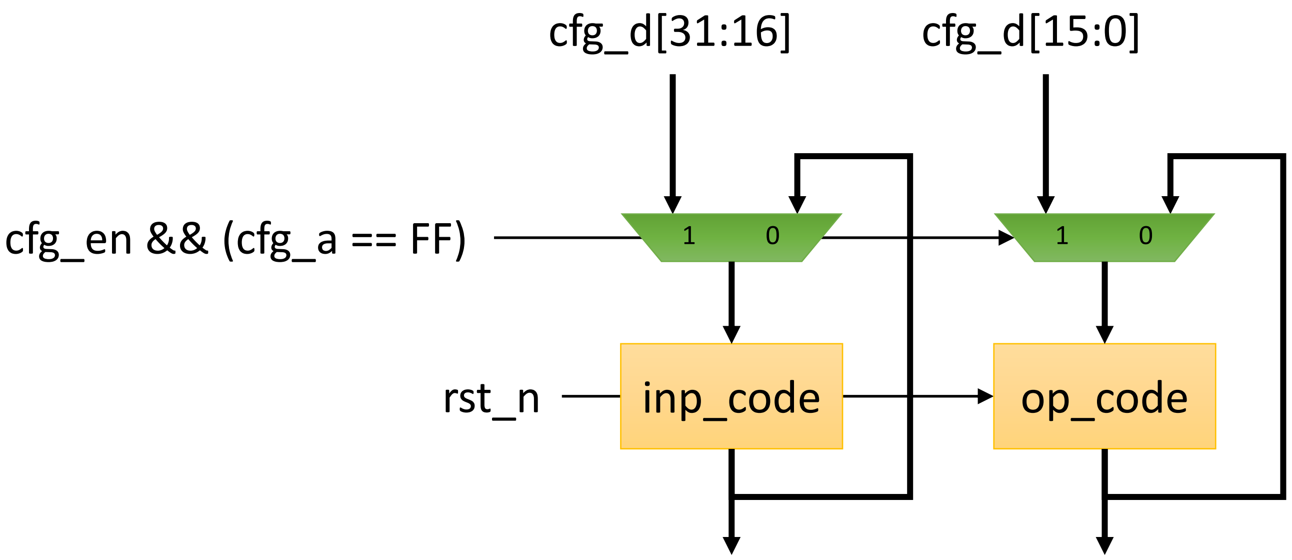
mode == bypass

Accumulation Operation

- For b, load signal can also be clk_en & en_acc (bit in the op code that enables accumulation)
- The data going into the register can also be 0/output of the PE after accumulation


ALU output
comp_res
0
To reset the accumulation register
0
1
en_acc & op_d_p
cfg_d[15:0]
0
1
(cfg_en && (cfg_a == F1))
data0
0
1
load = (cfg_en && (cfg_a == F1)) |
(clk_en & en_acc)
1
0
load or (mode == delay) or
((mode == valid) && clk_en)
reg
rst_n
1
0
mode == bypass

Configuration Registers in Diablo



inp_code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		mode (f)		mode (e)		mode (d)					mode (b)			mode (a)	

op_code

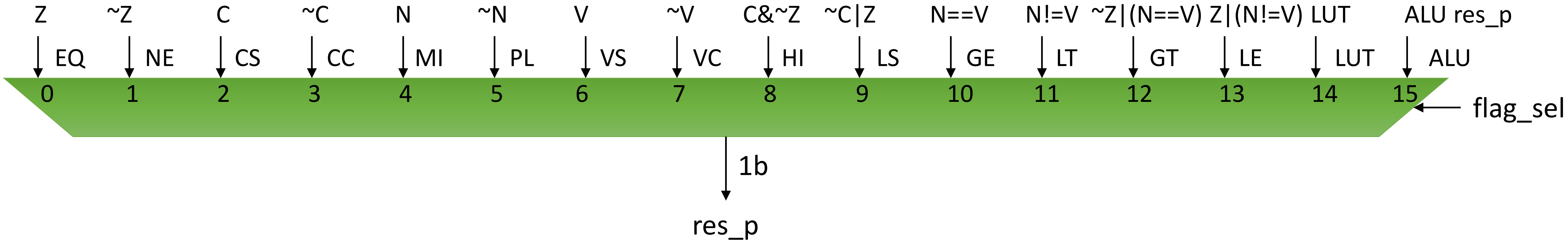
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flag_sel				irq_en		en_acc		pe_op_code							

Flag Logic

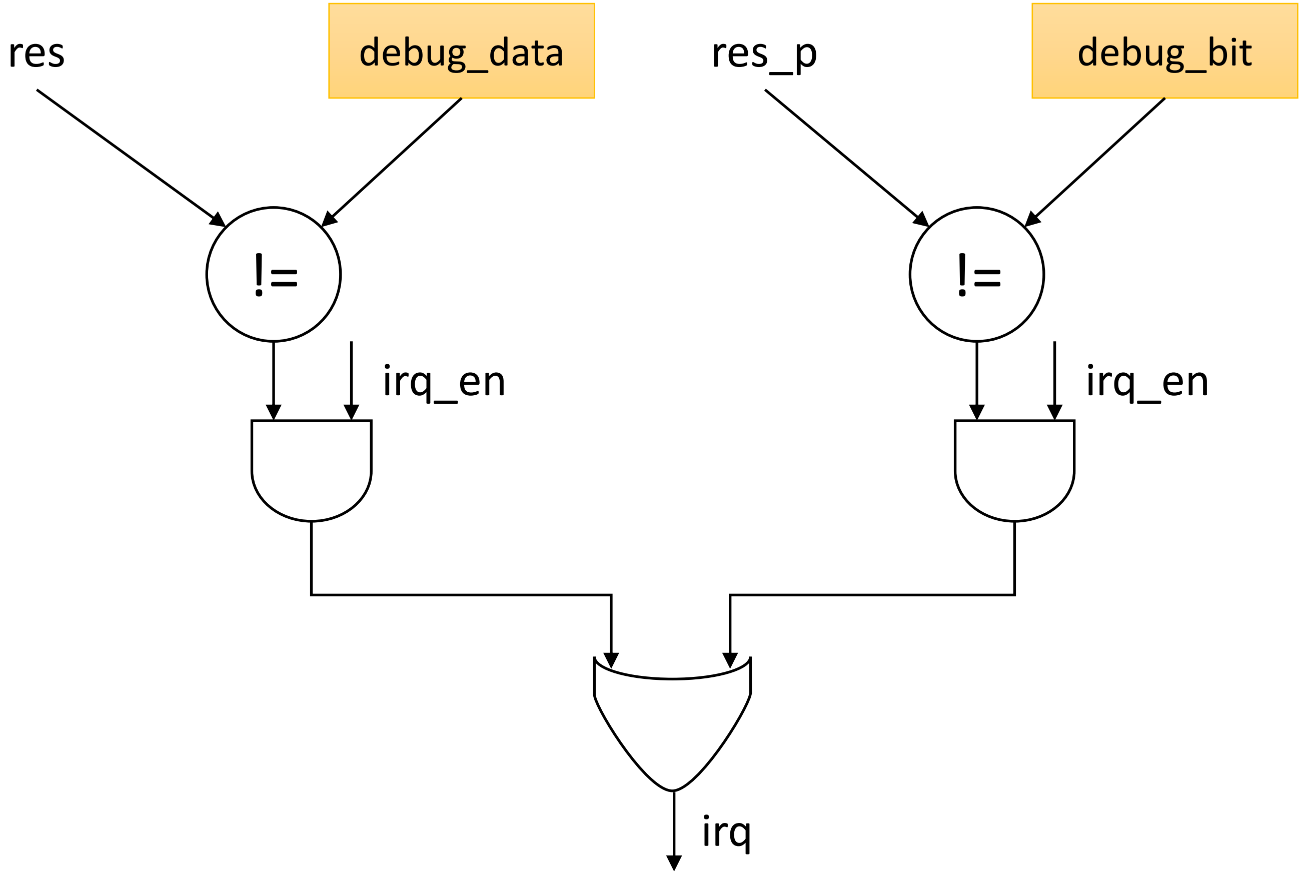
ALU produces

C = carry is generated
V = overflow
res 16b
res_p 1b

Z = res == 0
N = res is negative

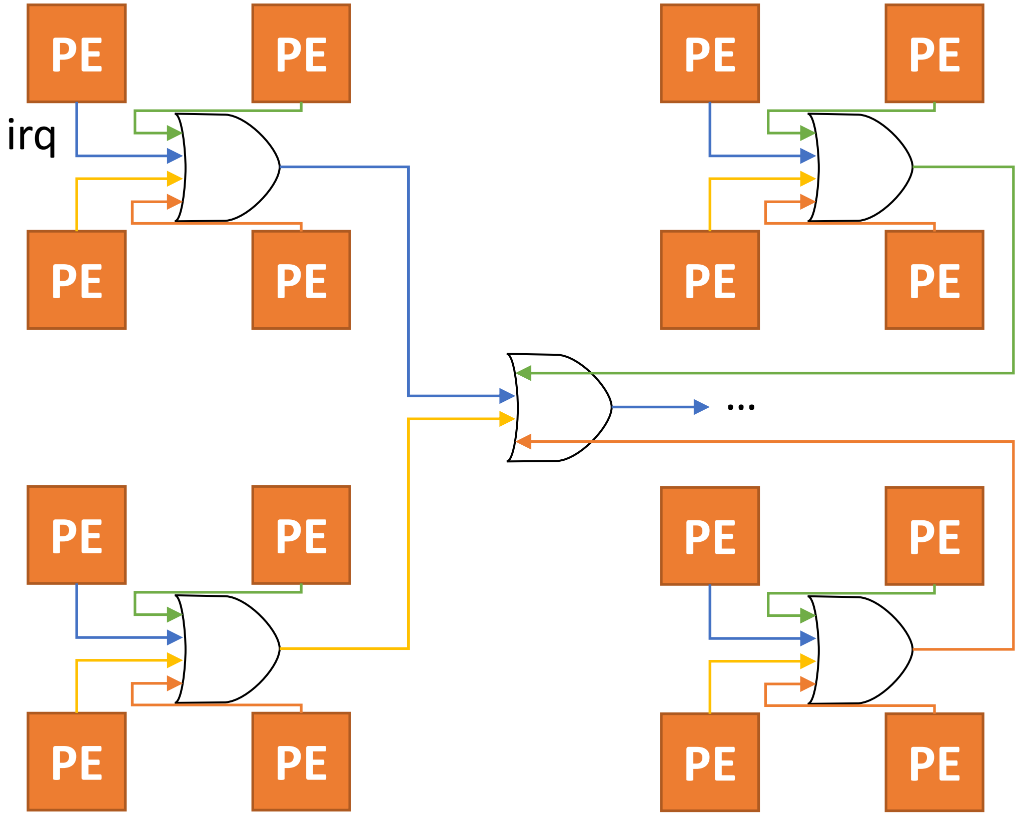


Breakpoint in Diablo

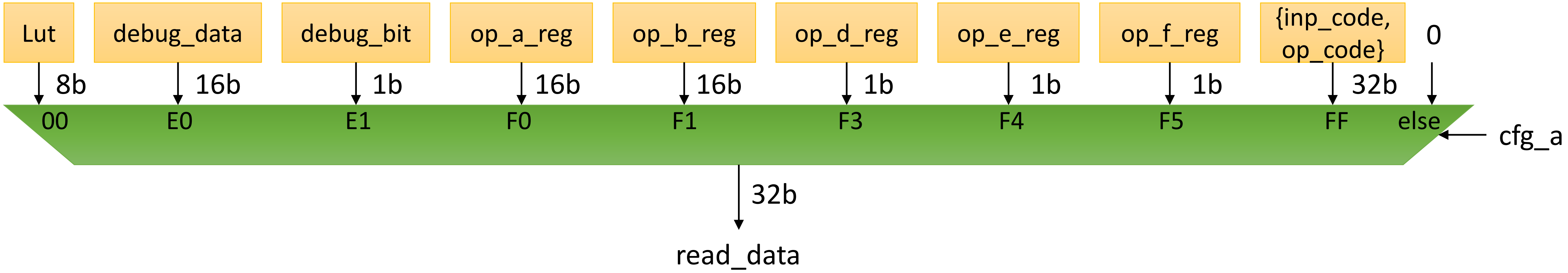


Global Signal Flow

- irq (interrupt request) signal is connected to global signal tiles (GST) in a hierarchical tree fashion



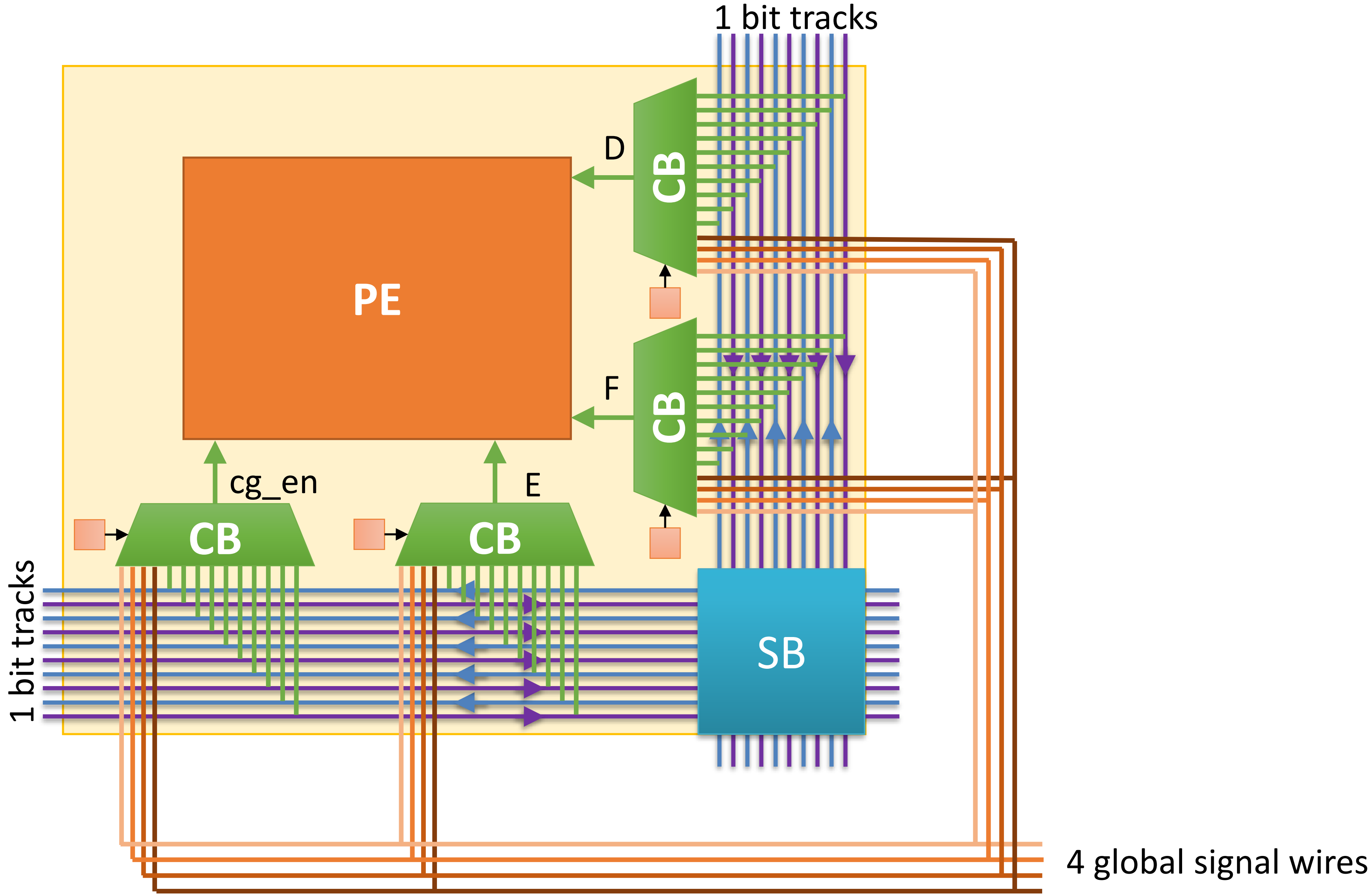
Reading Diablo registers in debug



Clock Gating in Diablo

- Each tile has 4 1 bit global signals coming in
- These go into all three 1 bit connection boxes
- There is another connection box that outputs the cg_en (clock gating enable) signal, reverse of which is the clk_en signal sent to the PE core
    - This connection box chooses this cg_en signal from 10 of the routing tracks on side 1 and the 4 global signals
    - Block diagram is on next slide

1 Bit Connection Boxes in Diablo



Improvements in Lassen over Diablo

- Lassen supports 16 bit Bfloat type
  - 1 b sign, 8 b exponent, 7 bit mantissa
  - Same range as Float32
- Bfloat add and multiply natively
- Micro-instructions to implement Bfloat transcendentals (divide, exp, power, log, sin, cos)
- Optimization passes (currently manual) to improve efficiency
- Proposed improvements : 16-bit LUT for masking operations
- Need review: Are the operations in lassen sufficient for implementing target applications

Diablo vs Lassen

Diablo Ops

Add
Sub
Abs
GTE_Max
LTE_Min
Sel
Mult0
Mult1
Mult2
SHR
SHL
Or
And
XOr

New Ops in Lassen

FPAdd
FPMult

FGetMant
FAddIExp
FSubExp
FCnvExp2F
FGetFInt
FGetFFrac

Micro-instructions

Other FP instructions requested by Jeff:
sub,
le lt, ge, gt, neq, neq,
neg, flr, ceil, abs, min, max, sqr

1. Allows easy porting of applications!
2. Enables new applications that need a large range like DNN training or complex ops like divide

Transcendentals Example - Divide

```
divident = Data(0x4288)  #68
divisor  = Data(0x4020)  #2.5
quotient = Data(0x41D9)  #68/2.5 = 27.2

mant          = pe_get_mant(asm.fgetmant(), divisor, Data(0))
lookup_result = mem_lut.div_lut(mant)
scaled_result = pe_scale_res(asm.fsubexp(), lookup_result, divisor)
quotient.     = pe_mult(asm.fp_mult(), scaled_result, divident)
```

Bfloat LUTs

```
class tlut:
   def div_lut(self, index):
     return {
     0   : 0x3f80,
     1   : 0x3f7e,
     2   : 0x3f7c,
      …
     126 : 0x3f01,
     127 : 0x3f00
     } [index]

   def ln_lut(self, index):
```

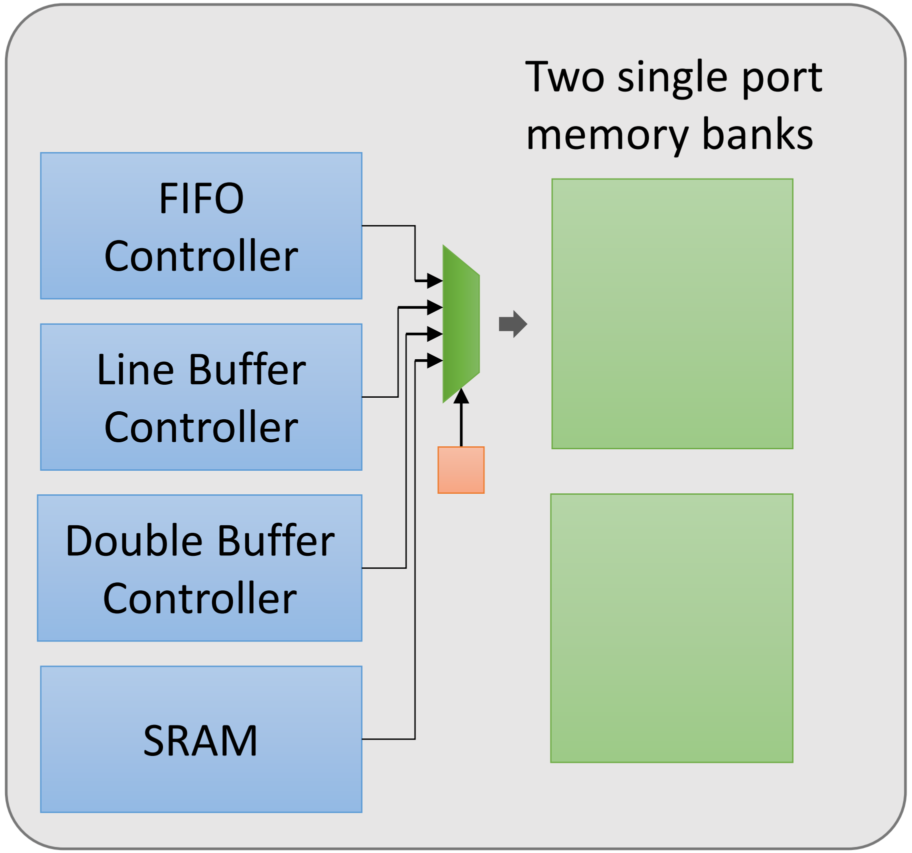
Memory

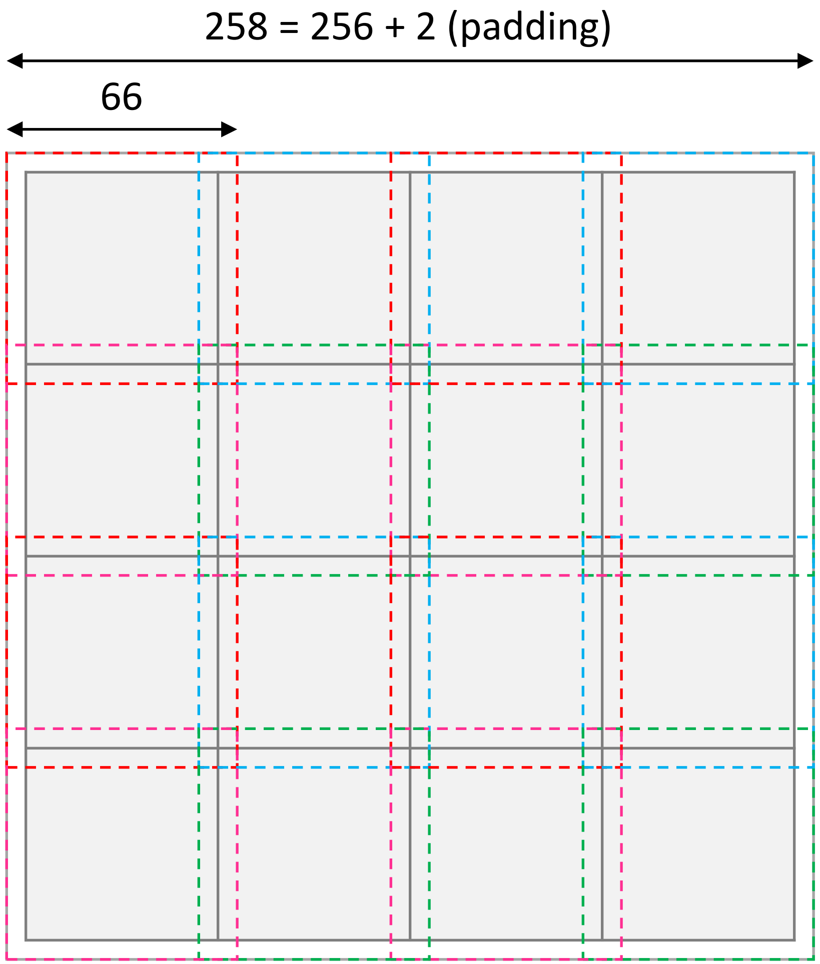
Max Strange

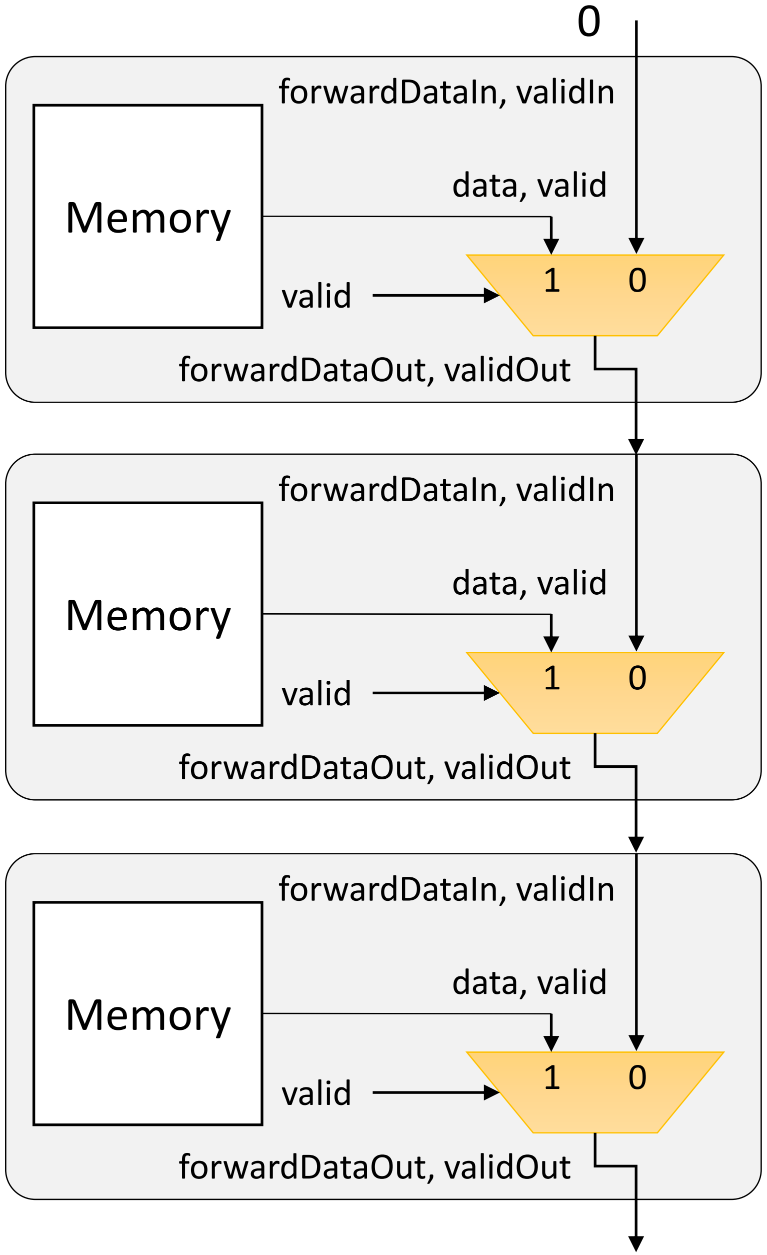
Memory Core in Jade

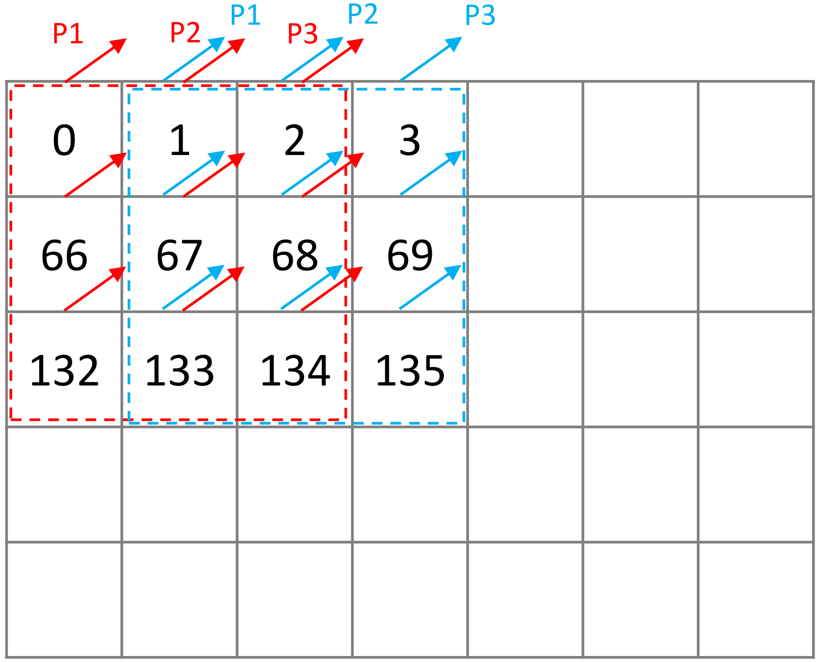
- There is a mux that chooses which one bit control signal goes out
  - 0, 1, 2, valid, full, almost_full
- Two banks both 16 bit wide and 512 entry
- There is a read and a write phase

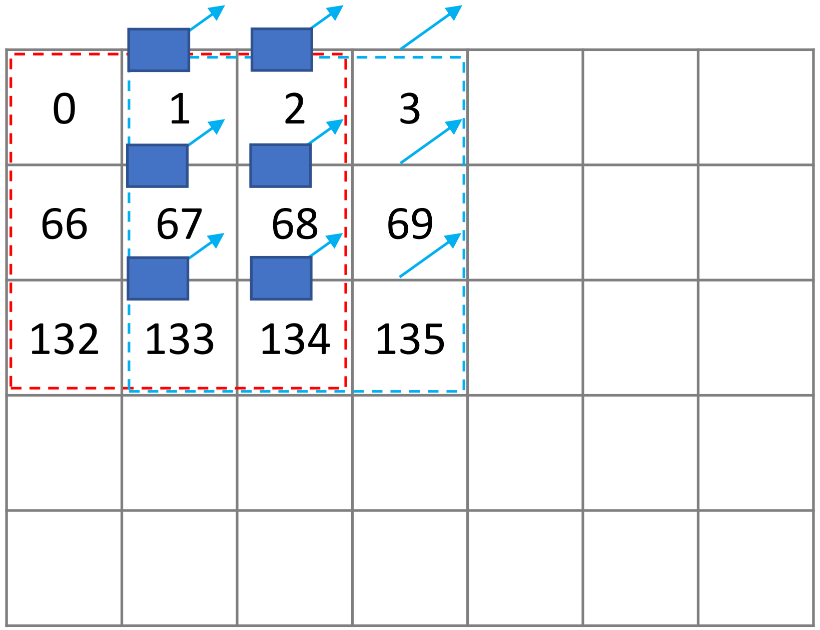
Memory


Two single port memory banks
FIFO Controller
Line Buffer Controller
Double Buffer Controller
SRAM


258 = 256 + 2 (padding)
66


0
forwardDataIn, validIn
Memory
data, valid
valid
1
0
forwardDataOut, validOut
forwardDataIn, validIn
Memory
data, valid
valid
1
0
forwardDataOut, validOut
forwardDataIn, validIn
Memory
data, valid
valid
1
0
forwardDataOut, validOut





9 output ports

Global Buffer

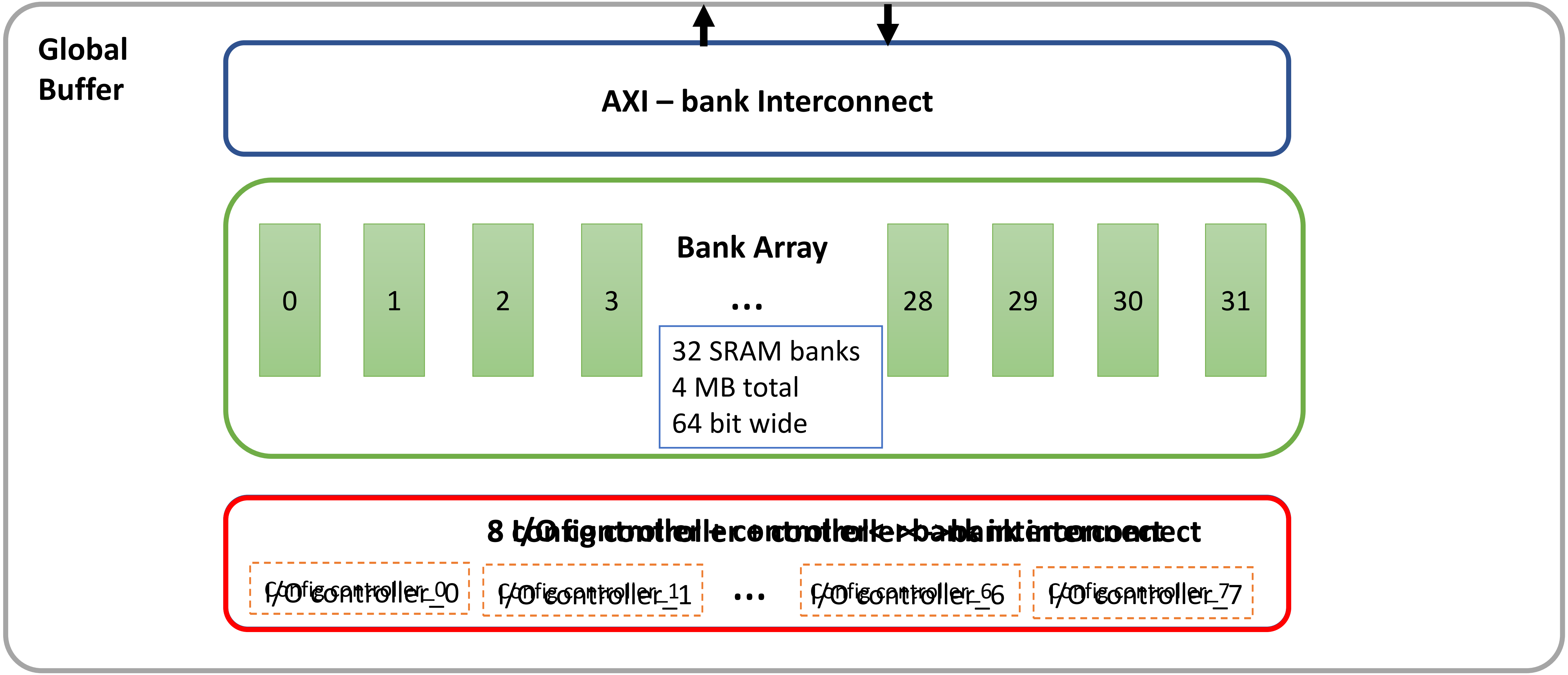
Taeyoung Kong

Global Buffer – Main Functions

- Highly banked buffer for **data reuse**
- Buffer to store bitstream for **parallel reconfiguration**

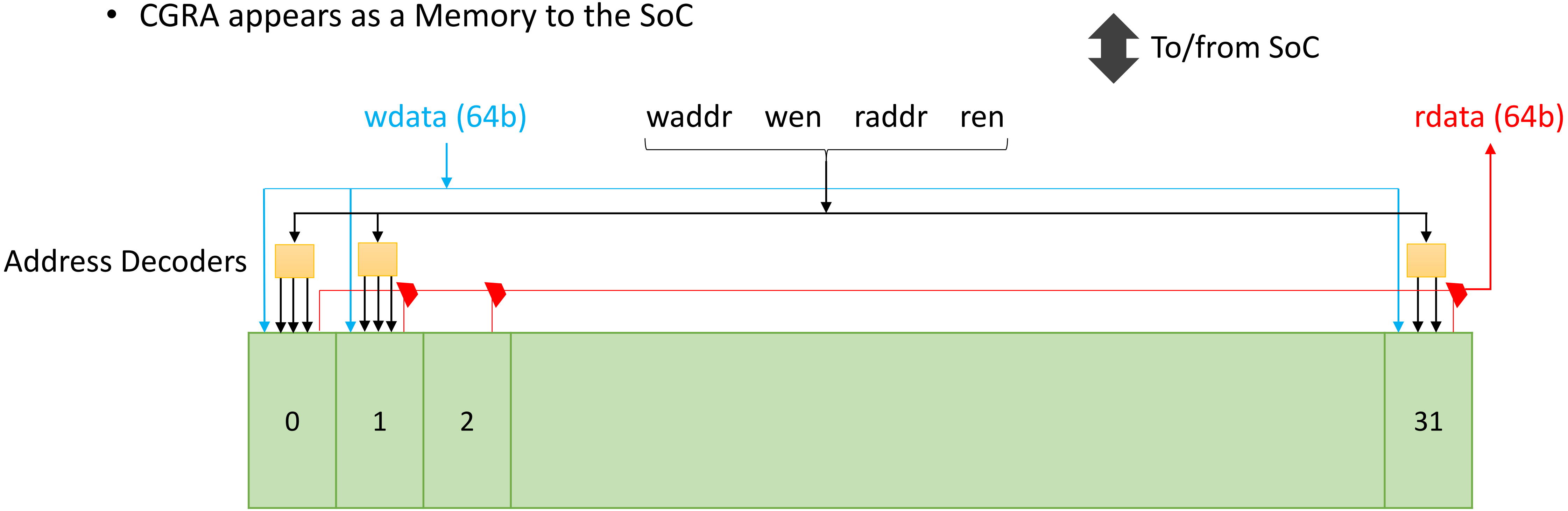
- Design Philosophy: Make global buffer simple and flexible

Global Buffer – Components



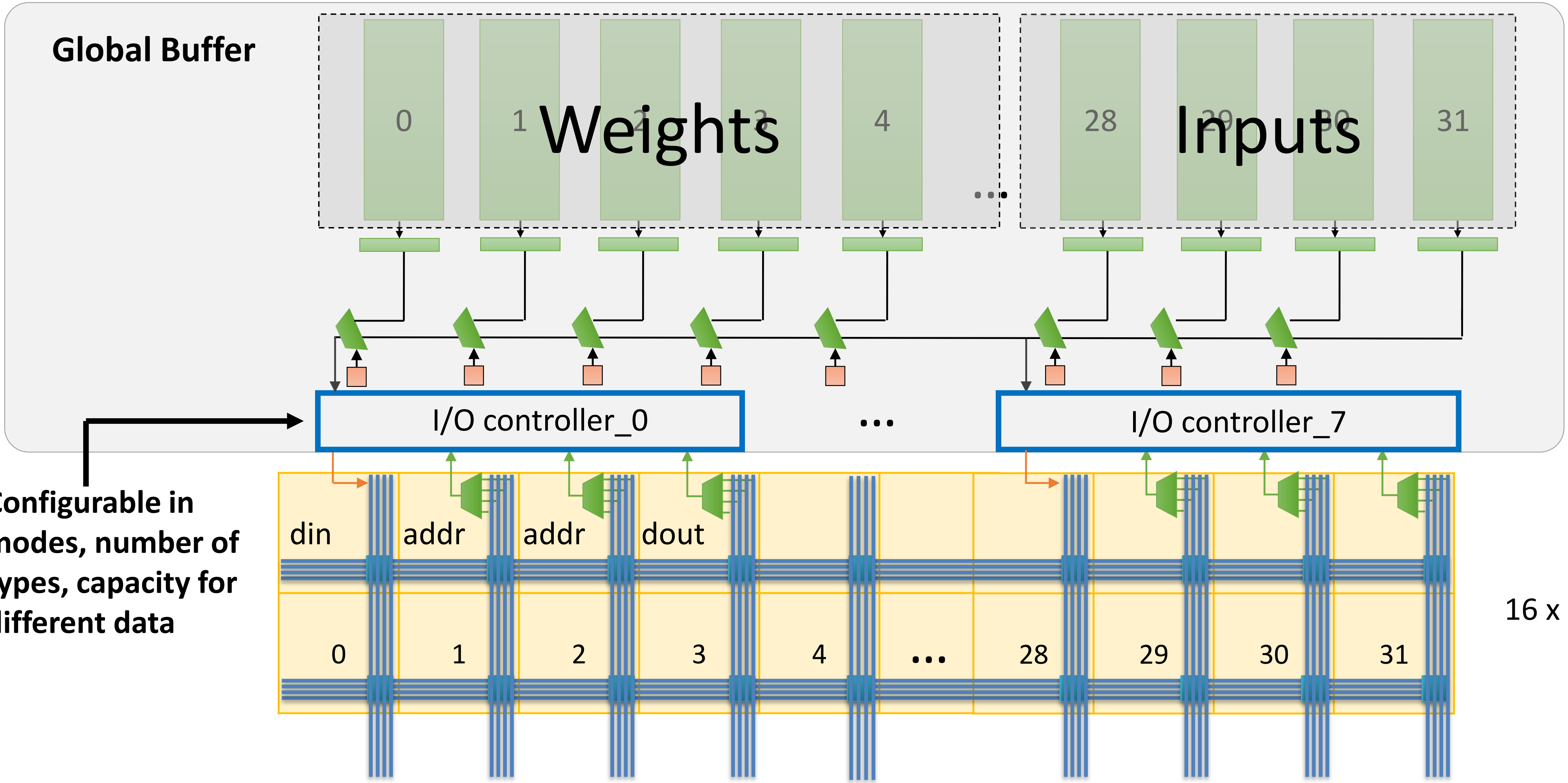
Global Buffer – SoC interface

- CGRA appears as a Memory to the SoC



- SoC provides a wrapper that converts between this simple interface into AXI interface which has block transfers
  - Block = 256 words * 64 bits/word
- Pipelined to support high clock frequency

Global Buffer – I/O controller

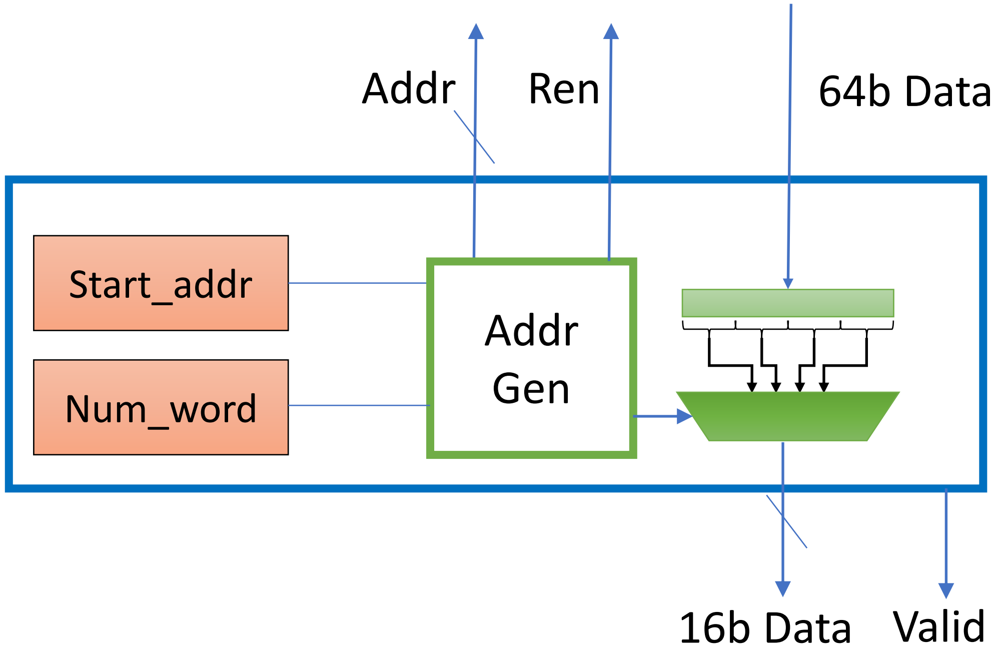


Configurable in modes, number of types, capacity for different data

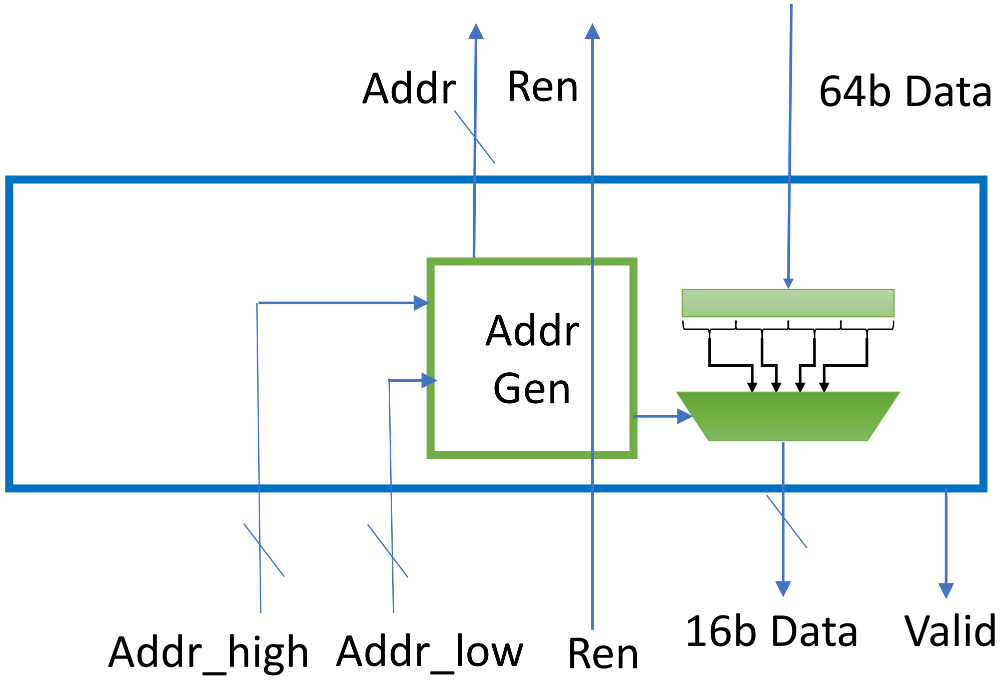
16 x 32 CGRA

Global Buffer – I/O controller

- To CGRA, we send out data and valid
- From CGRA, we get data and valid

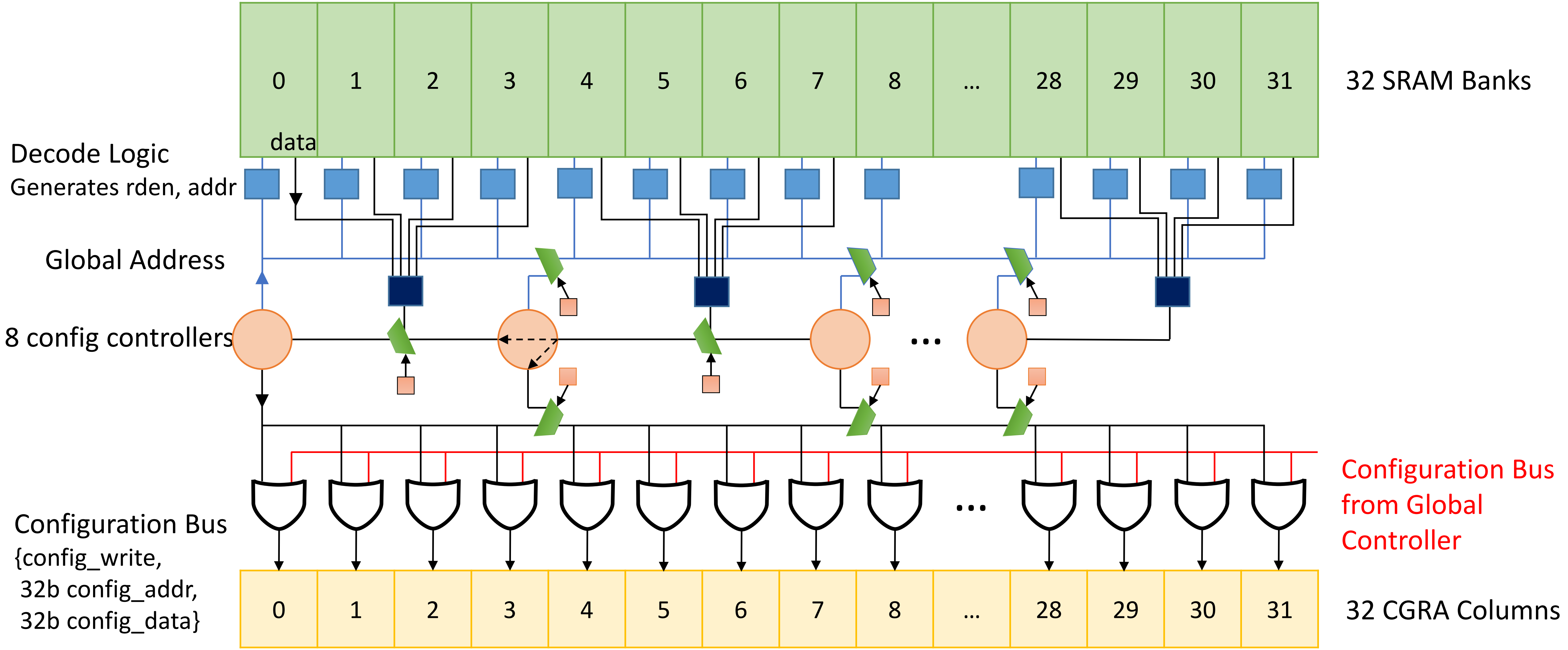


<FIFO mode>



<SRAM mode>

Global Buffer – Parallel Reconfiguration

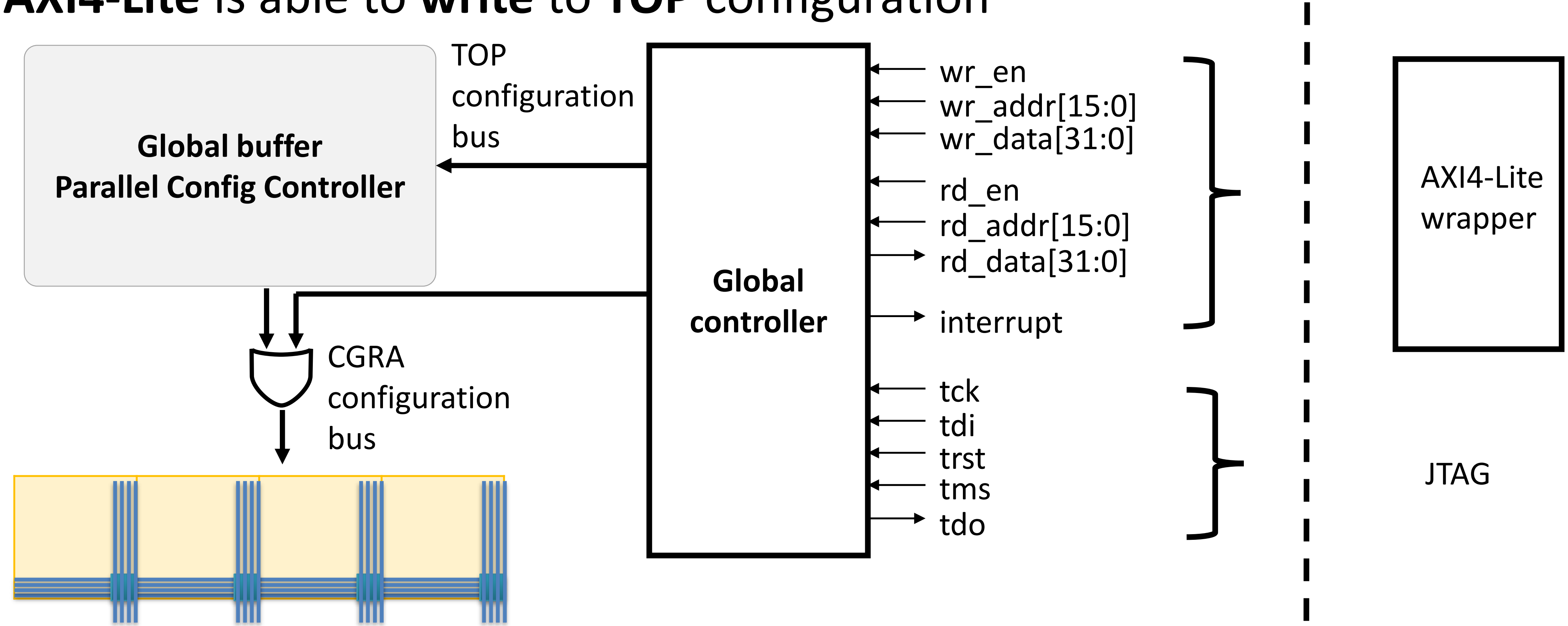


Global Controller

Taeyoung Kong & Alex Carsello

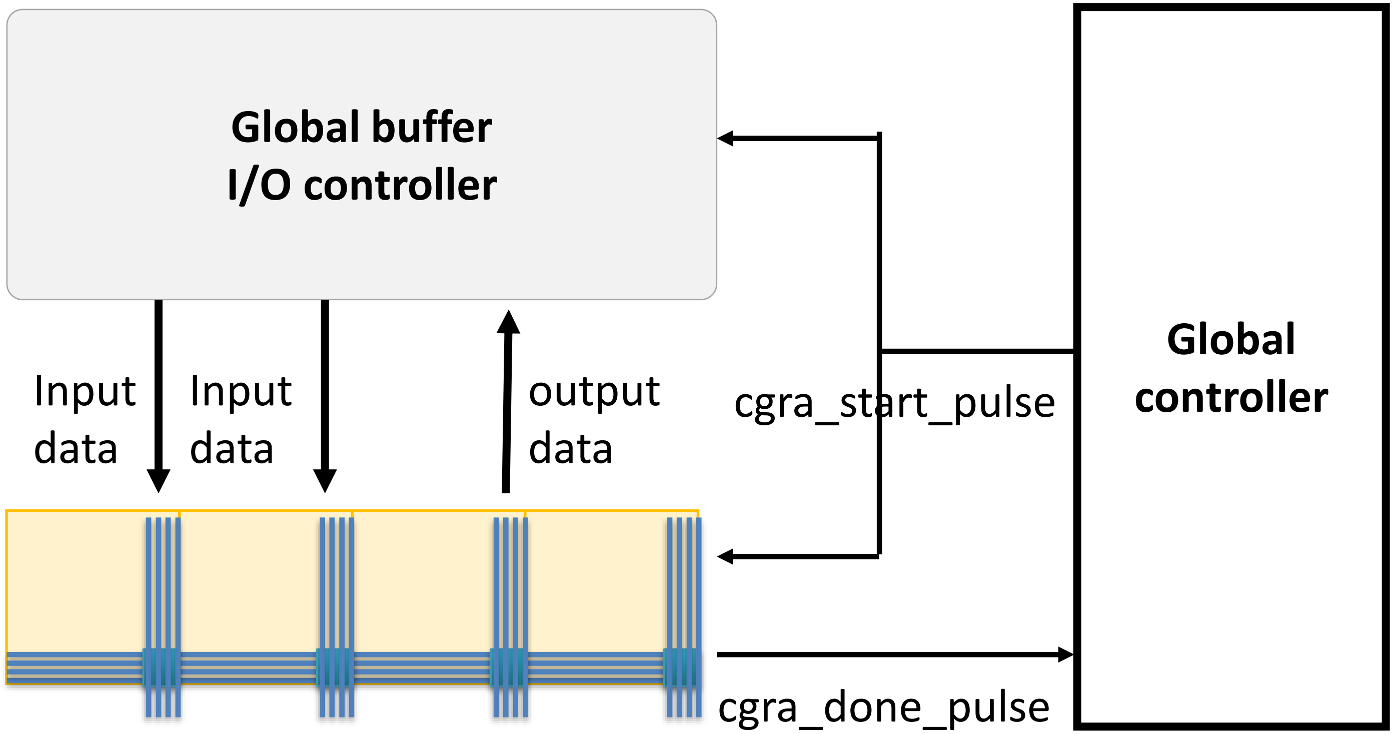
Configuration – AXI4-Lite & JTAG

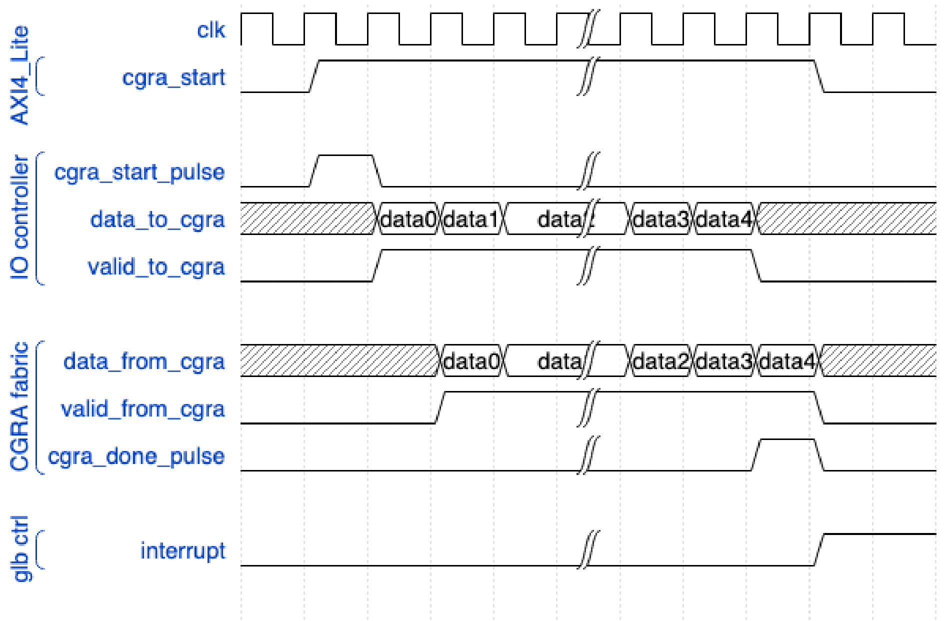
- **JTAG** is able to **write/read** to **CGRA** configuration and **TOP** configuration
- **AXI4-Lite** is able to **write** to **TOP** configuration



CGRA control – start & done

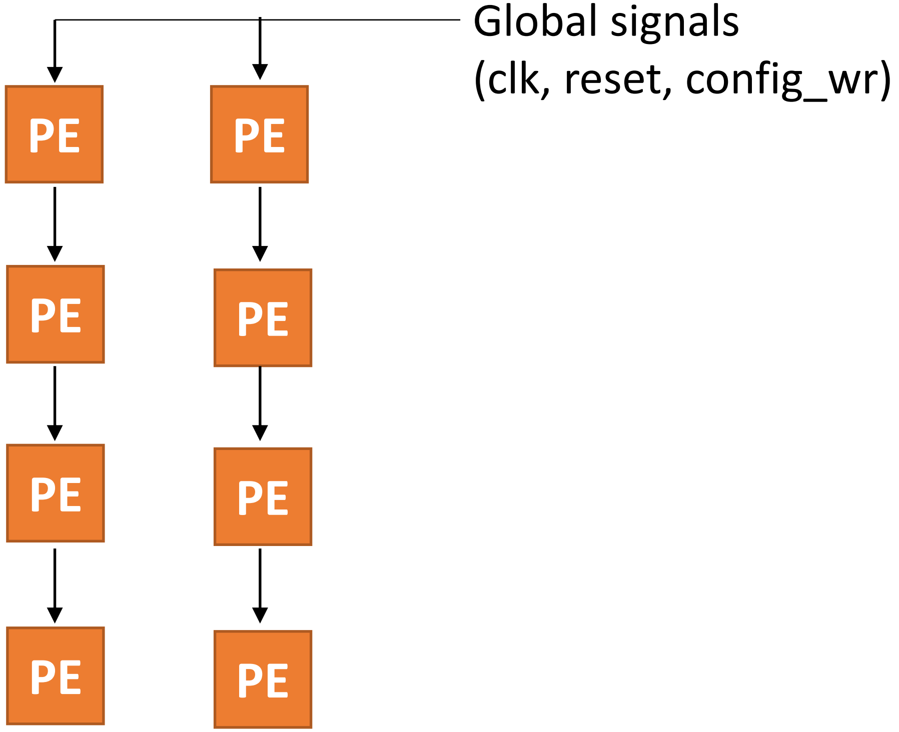
- **Block level** of application is controlled by **cgra_start** and **cgra_done**



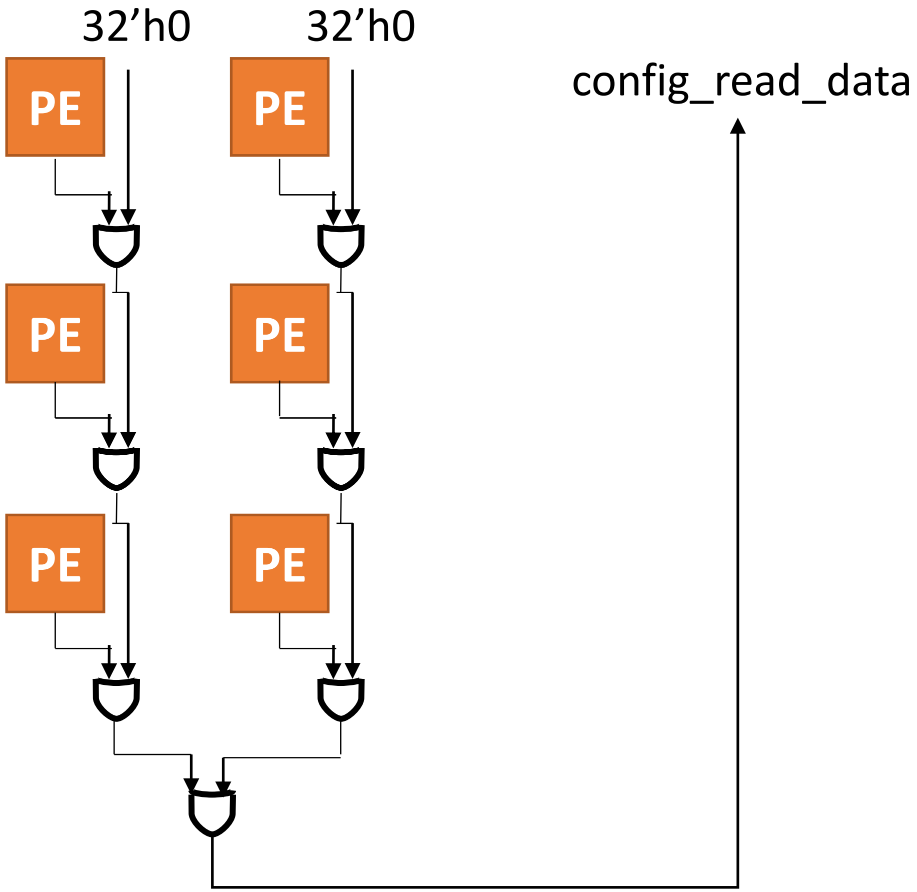


Global Signal Flow

<River-flowing input>



<River-flowing output>



Debug Support by JTAG

- JTAG can do everything!

- Debug CGRA fabric
  - Write/read CGRA-configuration registers
  - Write/read SRAM in CGRA fabric
  - 4bit stall signal
    - Every tile has configuration registers to select a stall signal from 4bits

- Debug CGRA fabric + global buffer
  - Write/read TOP-configuration registers
  - Write/read SRAM in global buffer
  - Write/read CGRA control register (cgra_start)

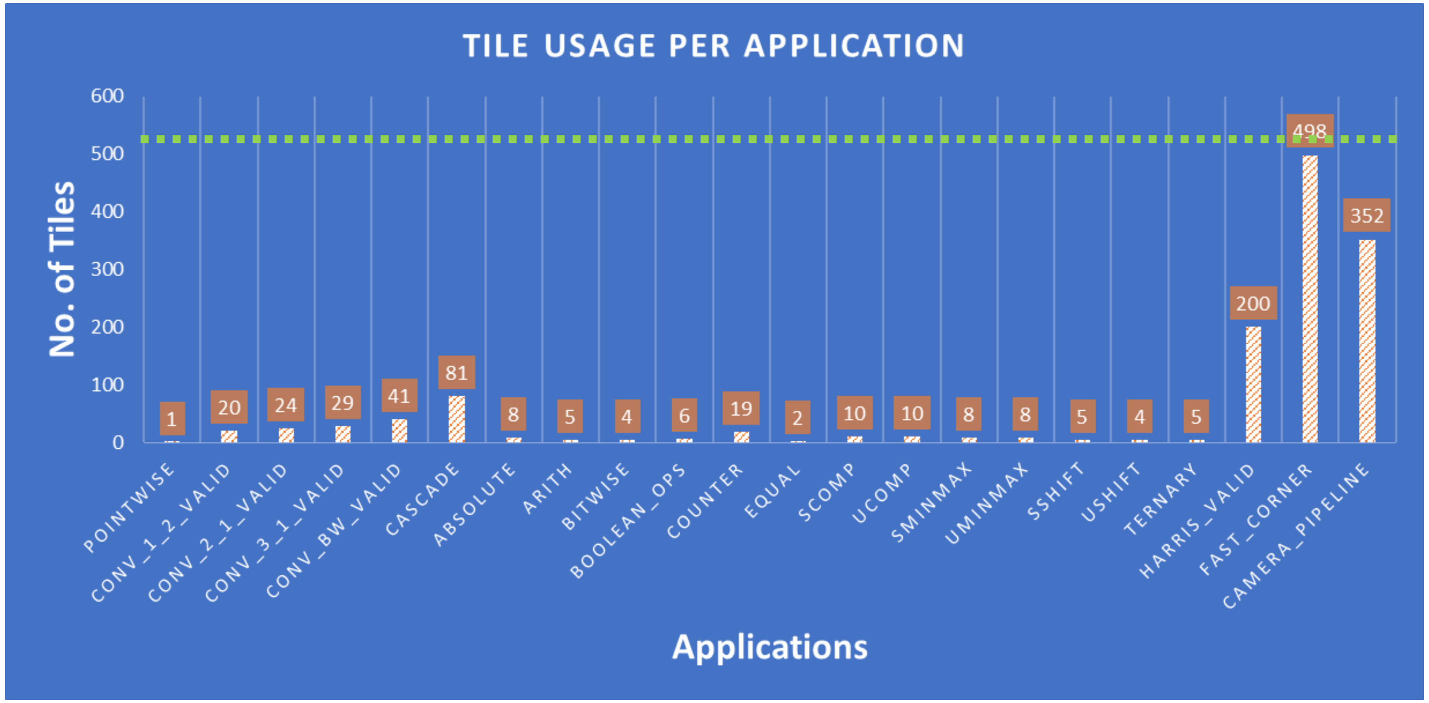
Power Domains

Ankita Nayak

Motivation

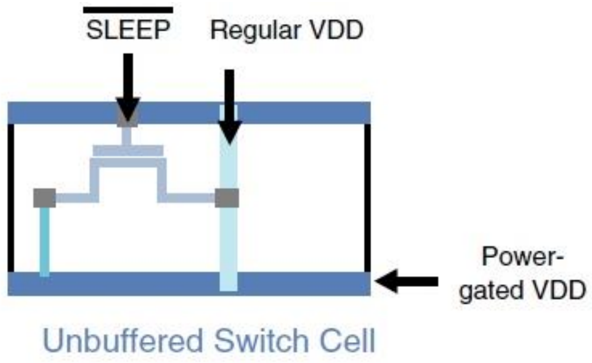
- Adoption of ‘Design for PPA’ mindset

- Save power by turning off tiles in the fabric not actively used

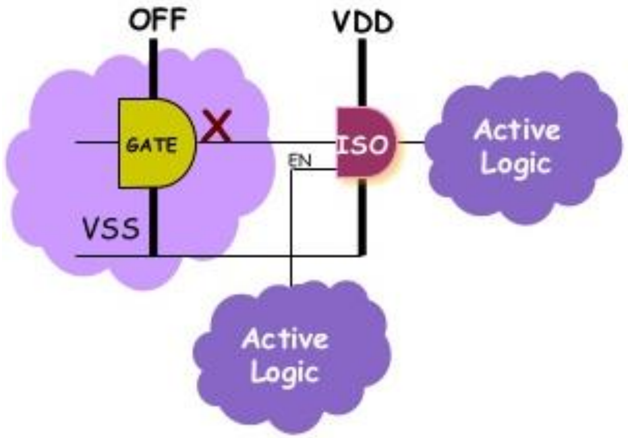


Architectural Features

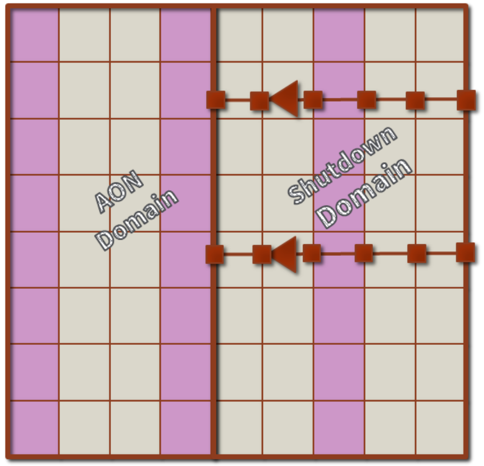
1. Power Switch Network Architecture



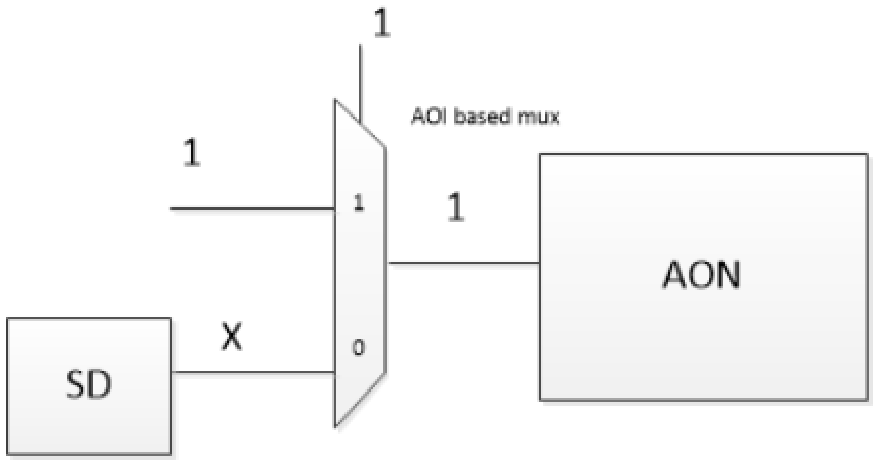
2. Boundary protection between the active and shutdown tiles



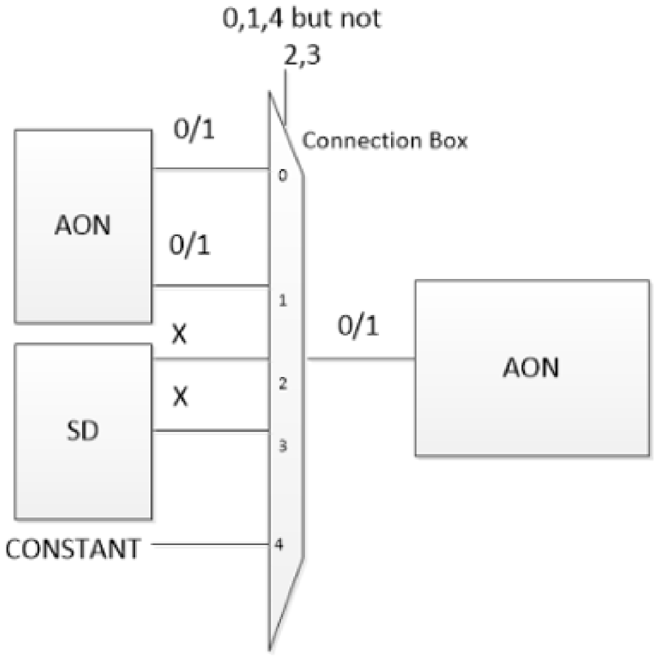
3. FT to connect IO near shutdown logic to the active part & for global signals



ON-OFF Boundary Protection



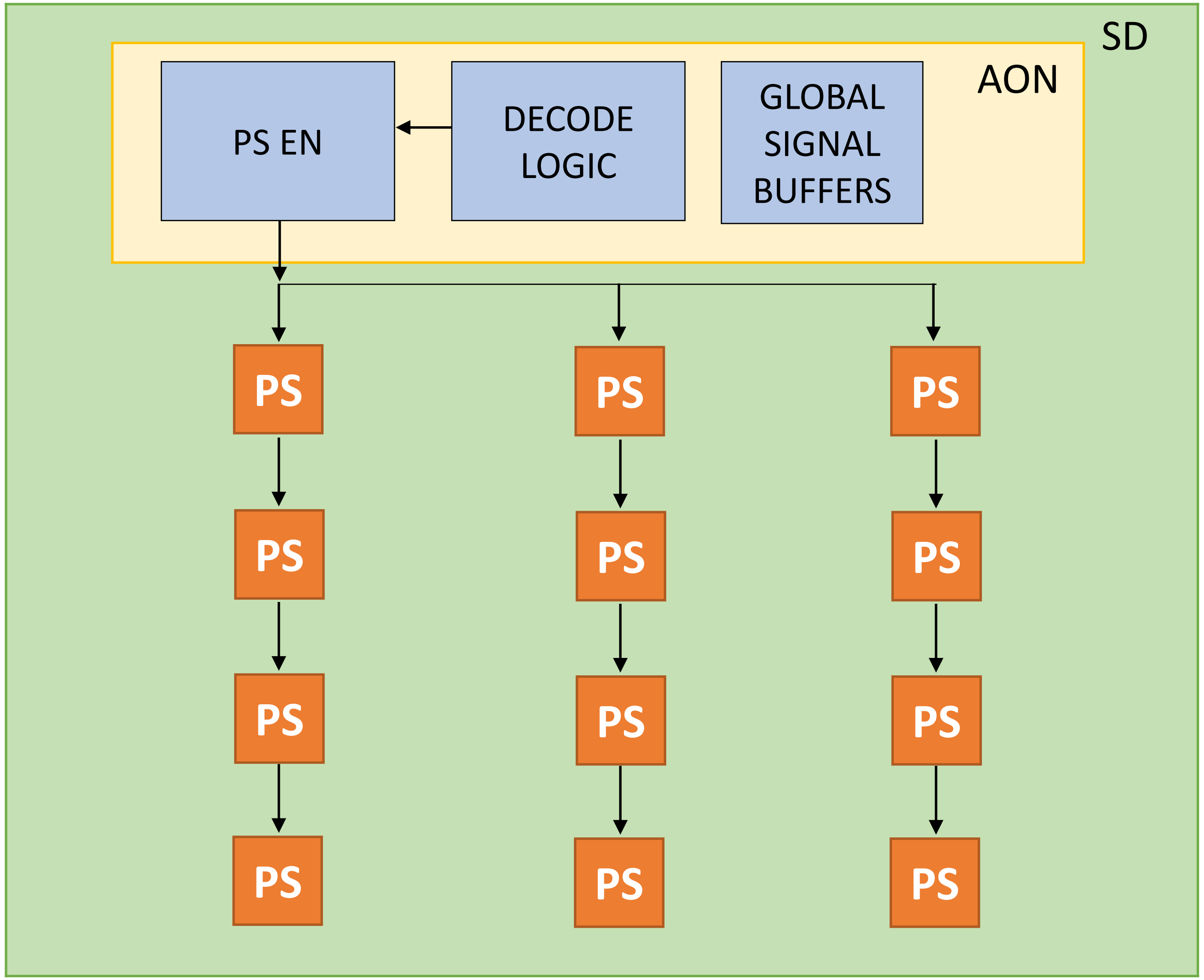
1. Redesign the muxes in the design to allow for inherent clamping mechanism



2. Use the existing muxes in the design for boundary protection

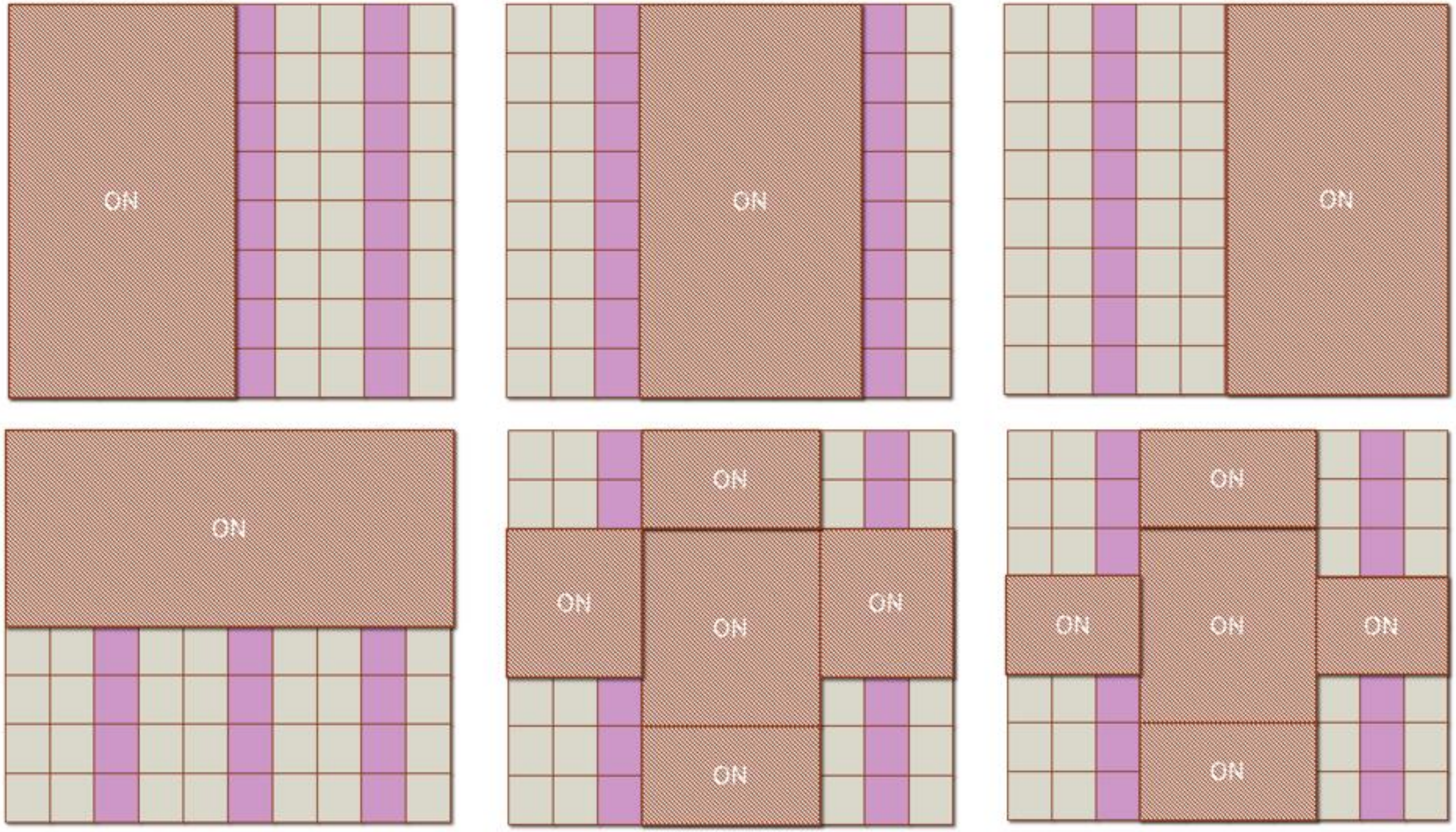
AON : Always ON
SD : Shutdown

Tile Architecture



- Configuration register and its decode logic for enabling the power switches is kept ON
- Buffers and logic on global signals are kept ON
- All other logic is shutdown when the tile is turned OFF

Top Level – Different sections can be turned ON/OFF by P&R



Summary of Garnet architectural features

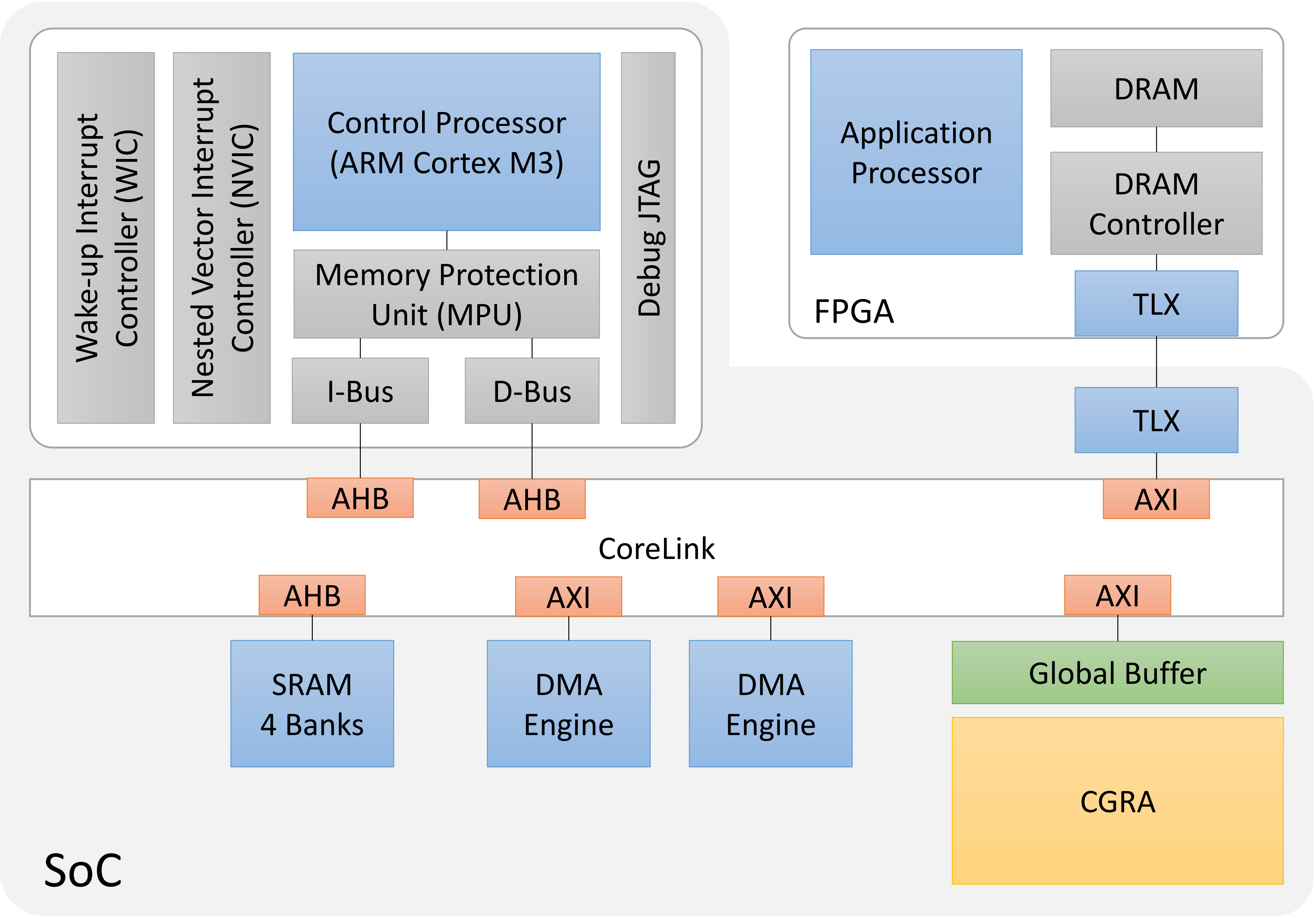
1. Support for Bfloat16, and for executing complex operations like divide using multiple PEs

2. Addition of a global buffer to create a memory hierarchy for efficiently executing neural networks

3. Fast reconfiguration support using global buffer and control processor

4. Addition of configurable power domains

SoC


Wake-up Interrupt Controller (WIC)
Nested Vector Interrupt Controller (NVIC)
Control Processor (ARM Cortex M3)
Memory Protection Unit (MPU)
I-Bus
D-Bus
Debug JTAG
Application Processor
DRAM
DRAM Controller
FPGA
TLX
TLX
AHB
AHB
AXI
CoreLink
AHB
AXI
AXI
AXI
SRAM 4 Banks
DMA Engine
DMA Engine
Global Buffer
CGRA
SoC