Designing CNN Accelerators
Day 2

Hyoukjun Kwon
(hyoukjun@gatech.edu)

Georgia Institute of Technology
Synergy Lab (http://synergy.ece.gatech.edu)

@SNU

Dec 27, 2017

Day 2 Agenda

- **BSV Sequential Logic implementation and execution model**
  - Memory Elements
  - Latency-Inter-module Communication
  - Modules with Multiple Rules
- **Traffic Patterns in CNN Accelerators**
  - Scatter
  - Gather
  - Local
- **Fixed Point Adder/Multiplier**

Memory Element Instantiation

- **Memory Elements as submodules**
  - Memory elements (register, FIFO) are implemented as independent modules
  - We instantiate memory elements as submodules
    - (ModuleInterfaceName) (user-defined module name) <- (ModuleName in implementation)
  - Ex)

```
Reg#(Bit#(16)) myReg <- mkReg(0);
```

A polymorphic Interface "Reg"

Load implenetation in module "mkReg"

Memory Elements in BSV

- **Register**
  - Initialization (module name)
    - mkReg(initial_value): Assign an initial value
    - mkRegU: Don’t assign an initial value
  - Operations
    - Read: multiple read within a cycle is allowed
    - Write (‘<=‘): only one write within a cycle is allowed
      written value is visible in the next cycle
  - Operation scheduling
    - Read < Write

Memory Elements in BSV

- **Register**
  - Example

```
Reg#(Bit#(4)) regA <- mkReg(2);
Reg#(Bit#(4)) regB <- mkRegU;
rule doExample;
  regA <= regA + 1;
  regB <= regA;
endrule
```

regA value is read twice
Written data is visible in the next cycle

Cycle	0	1	2	3	4
regA Value					
regB Value					

Memory Elements in BSV

- **FIFO (First-In-First-Out)**
  - Operations
    - enq: put a new element to the tail of a FIFO
    - deq: remove the head element (if exists)
    - first: returns the head element value (if exists)
    - notEmpty: returns true if the FIFO is not empty
  - Initialization
    - mkPipelineFifo: enq/first occurs after deq
    - mkBypassFifo: deq/first occurs after enq

Memory Elements in BSV

- **FIFO (First-In-First-Out)**
  - Declaration Syntax
    - **Fifo**#(Num_Elements, Types)

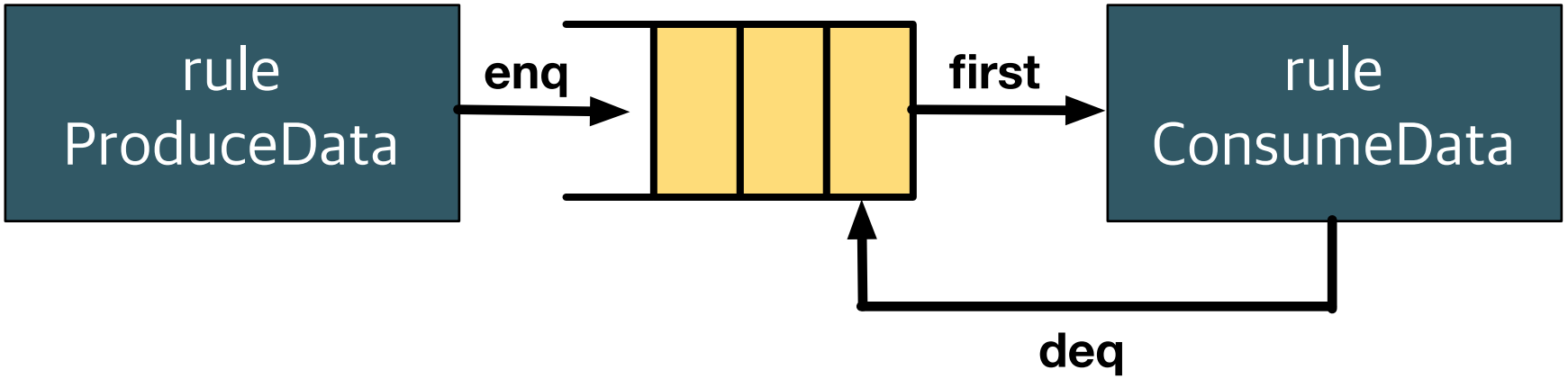
      user-defined_fifo_name <- (initilization)
    - Ex) **Fifo**#(3, **Bit**#(4)) myFifo <- mkPipelineFifo
  - Automatic rule/method stall
    - If a FIFO has no element and a rule tries to run ‘deq’ or ‘first’
    - If a FIFO is full and a rule tries to run ‘enq’

    * For both cases, the rule does not fire (execute) at that cycle

The stalled rule runs as soon as an element is enqued into the FIFO (for deq/first) or an element is dequed from the FIFO (for enq).

Memory Elements in BSV

- **FIFO (First-In-First-Out)**
  - Operation Example



Memory Elements in BSV

- **FIFO (First-In-First-Out)**
  - Operation Example1

```
Reg#(Bit#(16)) cycleReg <- mkReg(0);
Fifo#(2, Bit#(4)) fifoA <- mkPipelineFifo;

rule countCycles;
  cycleReg <= cycleReg + 1;
endrule

rule produceData;
  fifoA.enq(truncate(cycleReg));
endrule
...
```

Memory Elements in BSV

- **FIFO (First-In-First-Out)**
  - Operation Example1

```
rule consumeData;
  fifoA.deq; $display("Consumed %d", fifoA.first);
endrule
...
```

Cycle	0	1	2	3	4
fifoA.enq					
fifoA.first					
consumeData fire?					

Rule execution order: consumeData -> produceData

What happens when we use bypass FIFO?

Memory Elements in BSV

- **FIFO (First-In-First-Out)**
  - Operation Example2

```
Reg#(Bit#(16)) cycleReg <- mkReg(0);
Fifo#(2, Bit#(4)) fifoA <- mkBypassFifo;

rule countCycles;
  cycleReg <= cycleReg + 1;
endrule

rule produceData;
  fifoA.enq(truncate(cycleReg));
endrule
...
```

Memory Elements in BSV

- **FIFO (First-In-First-Out)**
  - Operation Example2

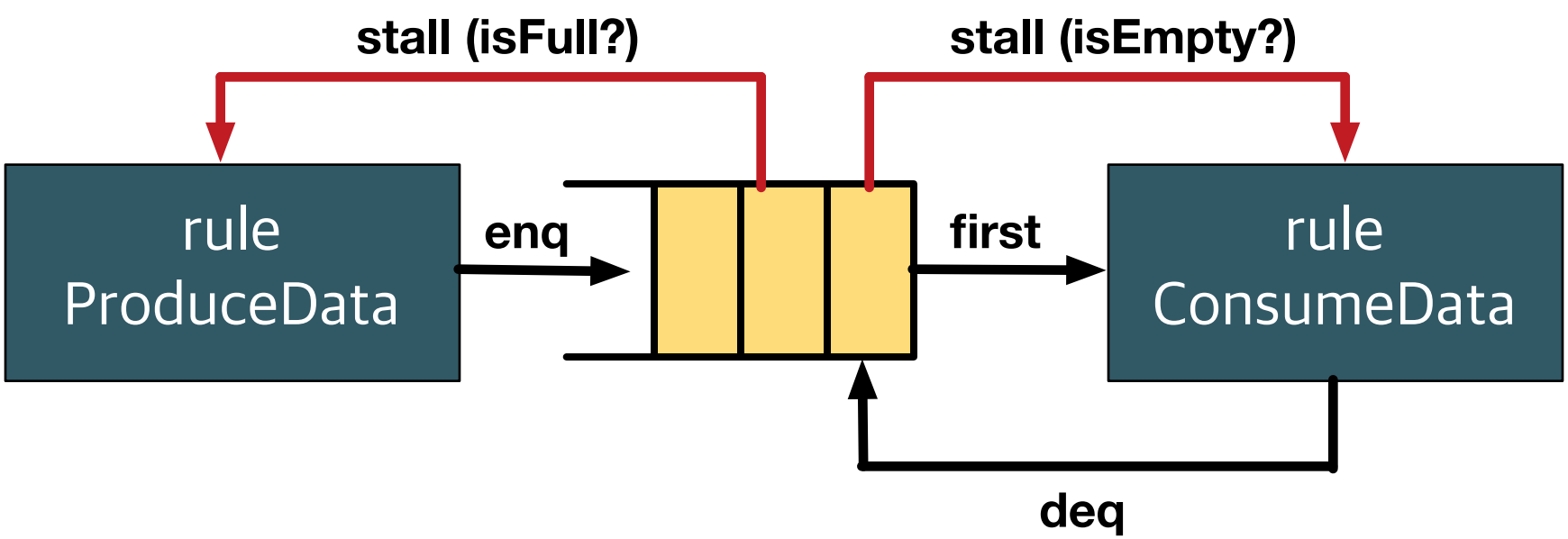
```
rule consumeData;
  fifoA.deq; $display("Consumed %d", fifoA.first);
endrule
...
```

Cycle	0	1	2	3	4
fifoA.enq					
fifoA.first					
consumeData fire?					

Rule execution order: produceData -> consumeData

Memory Elements in BSV

- **FIFO (First-In-First-Out)**
  - Operation Example



Implicit stall control based on FIFO occupancy

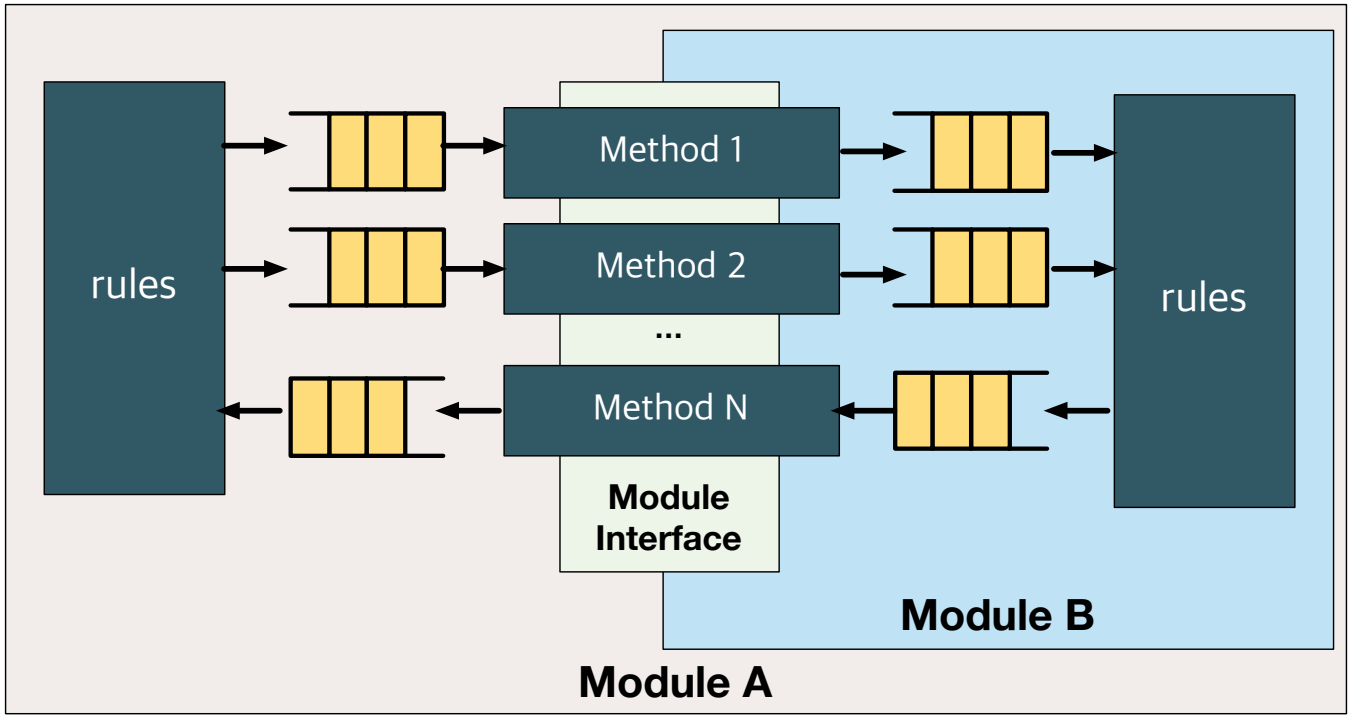
Enables "latency insensitive inter-module communication"

Day 2 Agenda

- **BSV Sequential Logic implementation and execution model**
  - Memory Elements
  - Latency-Inter-module Communication
  - Modules with Multiple Rules
- **Traffic Patterns in CNN Accelerators**
  - Scatter
  - Gather
  - Local
- **Fixed Point Adder/Multiplier**

LI Inter-Module Communication

- **Latency-insensitive (LI) inter-module communication model**



Rules wait until (1) all the necessary data is in input FIFOs and (2) at least one slot of output FIFO is available

Why is it good?

Module Interface and Methods

- **Defining an interface (syntax)**

```
// interface definition
interface (Interface_Name);
  // method definition
  method (return_type) (method_name) (arguments);
  // an interface can contain multiple methods
endinterface
```

Module Interface and Methods

- **Example**

```
interface ALU;
  method Action putArguments(OpCode newOp,
                             Word newArgA, Word newArgB);
  method ActionValue#(Word) getResults;
  method Bool isInitialized;
endinterface
```

Action method: Similar to "void" in C. Involves state updates (register, FIFO, etc.)

ActionValue#(T) method: Involves state updates (register, FIFO, etc.) + returns a value with type T

Module Interface and Methods

- **Implementing an interface – example**

```
module mkExampleModule(ALU);
    // module implementations (omited)
    //....

    method Action putArguments(OpCode newOp,
                               Word newArgA, Word newArgB);
     opCode <= newOp; //....
    endmethod

    method ActionValue#(Word) getResults;
     isValidArgs <= False; return res;
    endmethod

    method Bool isInitialized = inited;

endmodule
```

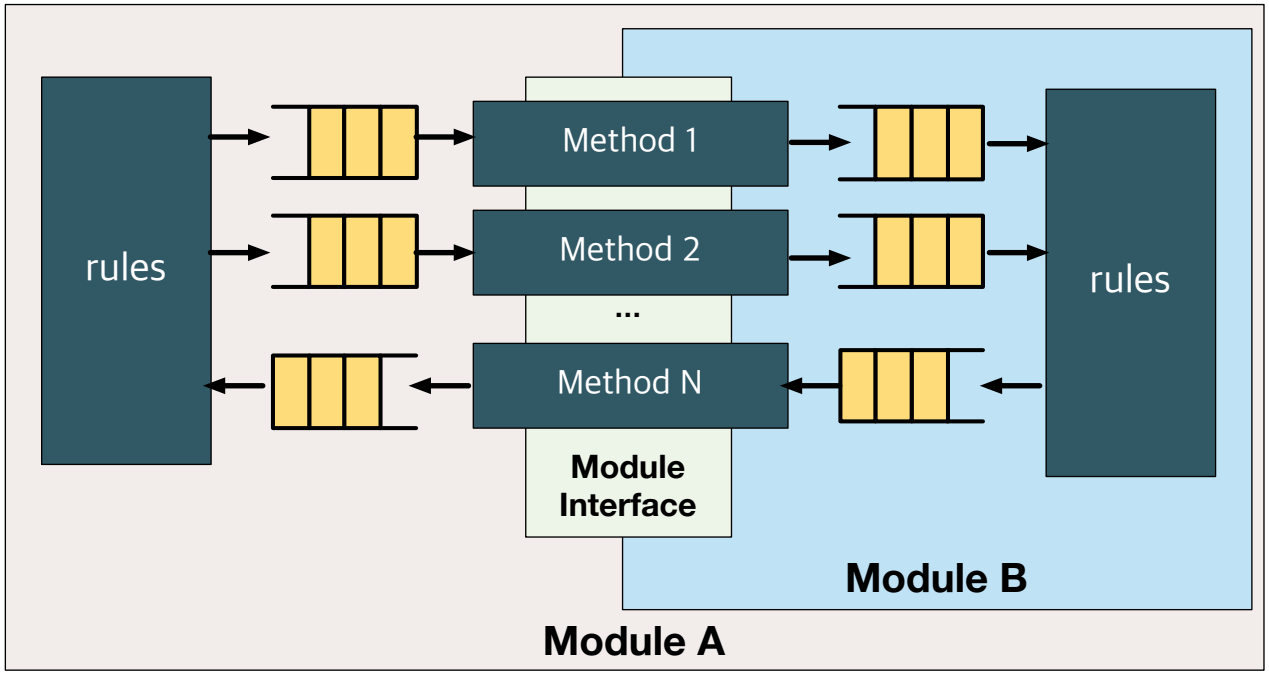
state update

returns a value

return values can also be described in this manner

LI Inter-Module Communication

- **Implementations**



(1) methods just enque data to input FIFOs and deque from output FIFOs

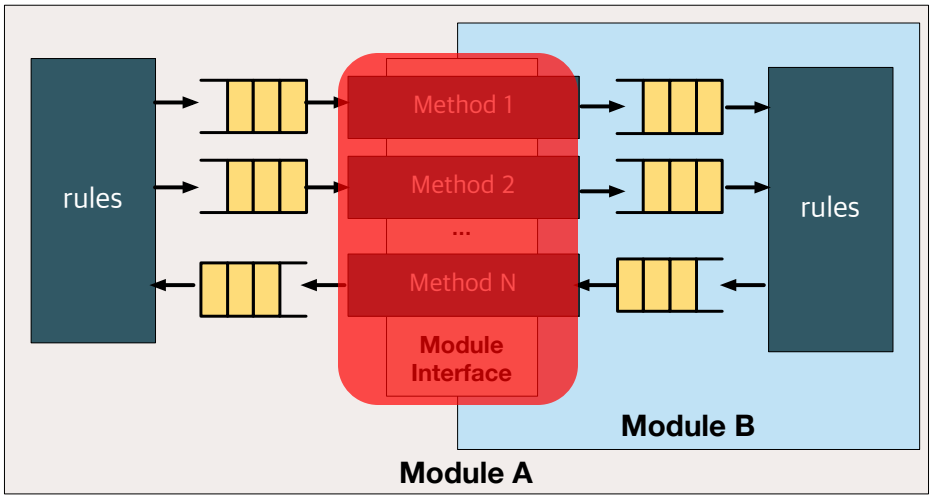
(2) rules deq input values from input FIFOs and enq output values to output FIFOs

LI Inter-Module Communication

- **Implementation Example**

```
interface ModuleBIfc;
  method Action sendData(Bit#(16) newData);
  method ActionValue#(Bit#(16)) getData;
endinterface
```

Required. Why?



LI Inter-Module Communication

- **Implementation Example**

```
module mkModuleB(ModuleBIfc);
  Fifo#(2, Bit#(16)) inputFifo <- mkPipelineFifo;
  Fifo#(2, Bit#(16)) outputFifo <- mkPipelineFifo;

  rule incValue;
    let data = inputFifo.first; inputFifo.deq;
    outputFifo.enq(data+1);
  endrule

  method Action sendData(Bit#(16) newData);
    inputFifo.enq(newData);
  endmethod

  method ActionValue#(Bit#(16)) getData;
    outputFifo.deq; return outputFifo.first;
  endmethod

endmodule
```

Day 2 Agenda

- **BSV Sequential Logic implementation and execution model**
  - Memory Elements
  - Latency-Inter-module Communication
  - Modules with Multiple Rules
- **Traffic Patterns in CNN Accelerators**
  - Scatter
  - Gather
  - Local
- **Fixed Point Adder/Multiplier**

Modules with Multiple Rules

- **Rule Scheduling**
  - Rules are fundamental atomic unit of hardware behavior in BSV
    - **[All-or-Nothing]** Run entire statements in a rule. If at least one of the statements cannot be executed at a certain cycle (e.g., enq to a full FIFO), the rule stalls.
  - BSV scheduler tries to execute as many rules an possible in parallel
  - Executing all the rules might not be possible

When?

Modules with Multiple Rules

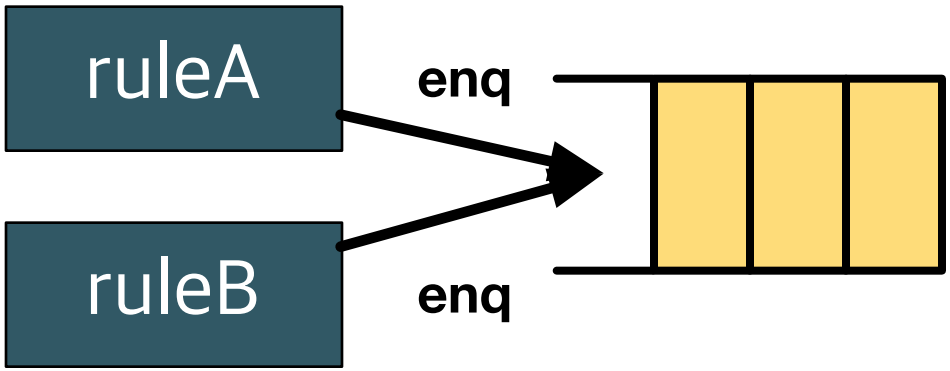
- **Rule conflict**

```
rule incValue;
    let data = inputFifo.first; inputFifo.deq;
    outputFifo.enq(data+1);
endrule

rule decValue;
    let data = inputFifo.first; inputFifo.deq;
    outputFifo.enq(data-2);
endrule
```

What happens?

Modules with Multiple Rules

- **Rule conflict**



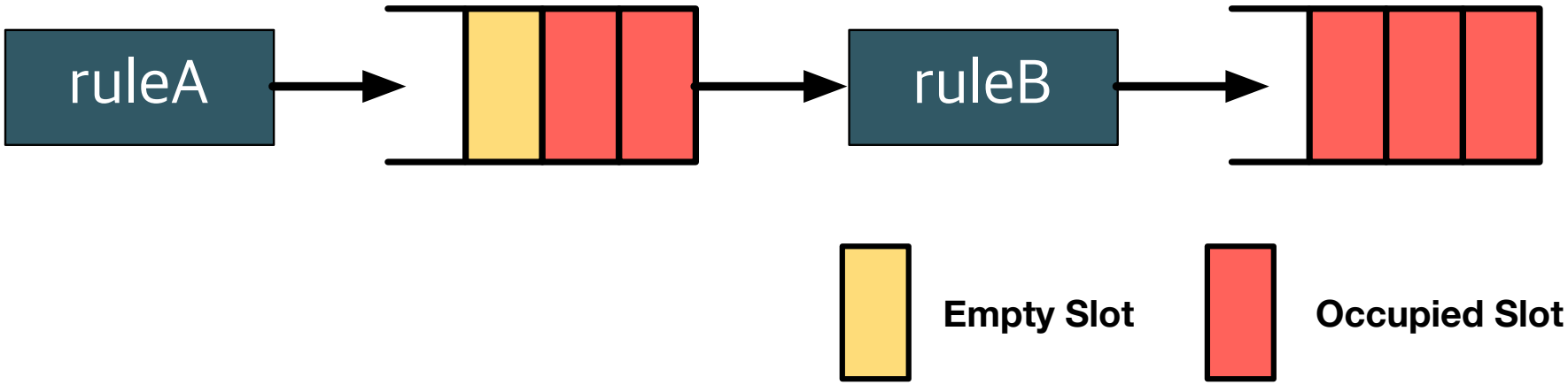
Resource Conflict
(Similar to Structural Hazard)

Although both ruleA and ruleB are ready to fire, only one of them can fire each cycle.

Each method in an interface can be called only once at each cycle

Modules with Multiple Rules

- **Independent scheduling**



RuleB cannot fire beacuse its output FIFO is full

Although ruleB cannot fire, ruleA can fire.

Modules with Multiple Rules

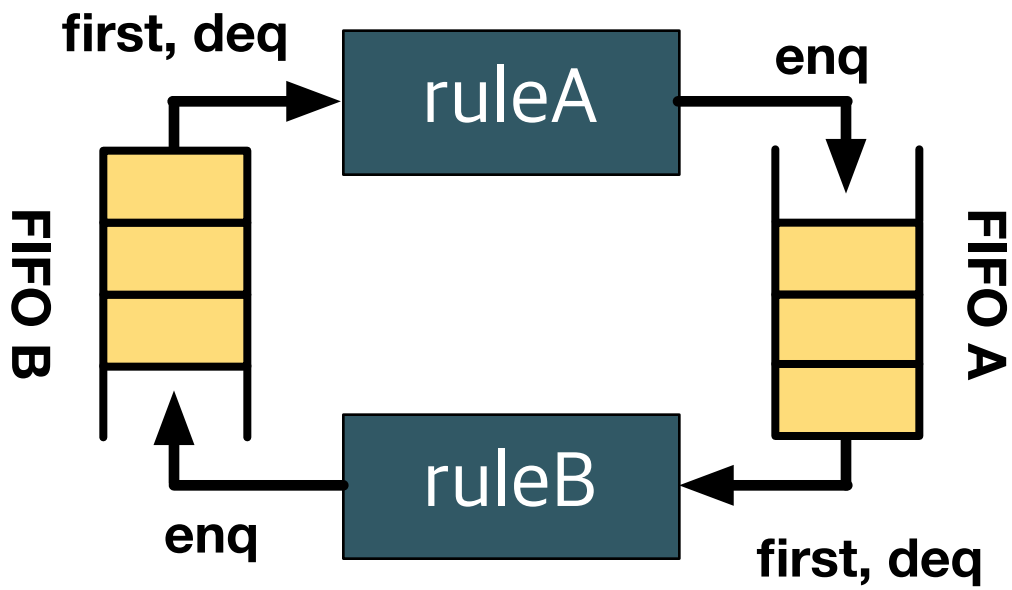
- **Cyclic dependence**

```
Fifo#(2, Bit#(16)) fifoA <- mkBypassFifo;
Fifo#(2, Bit#(16)) fifoB <- mkBypassFifo;

rule ruleA;
    let data = fifoB.first; fifoB.deq;
    fifoA.enq(data-1);
    outputFifo.enq(data-1);
endrule

rule ruleB;
    let data = fifoA.first; fifoA.deq;
    fifoB.enq(data+1);
endrule
```

Any problem?

Modules with Multiple Rules

- **Cyclic dependence**

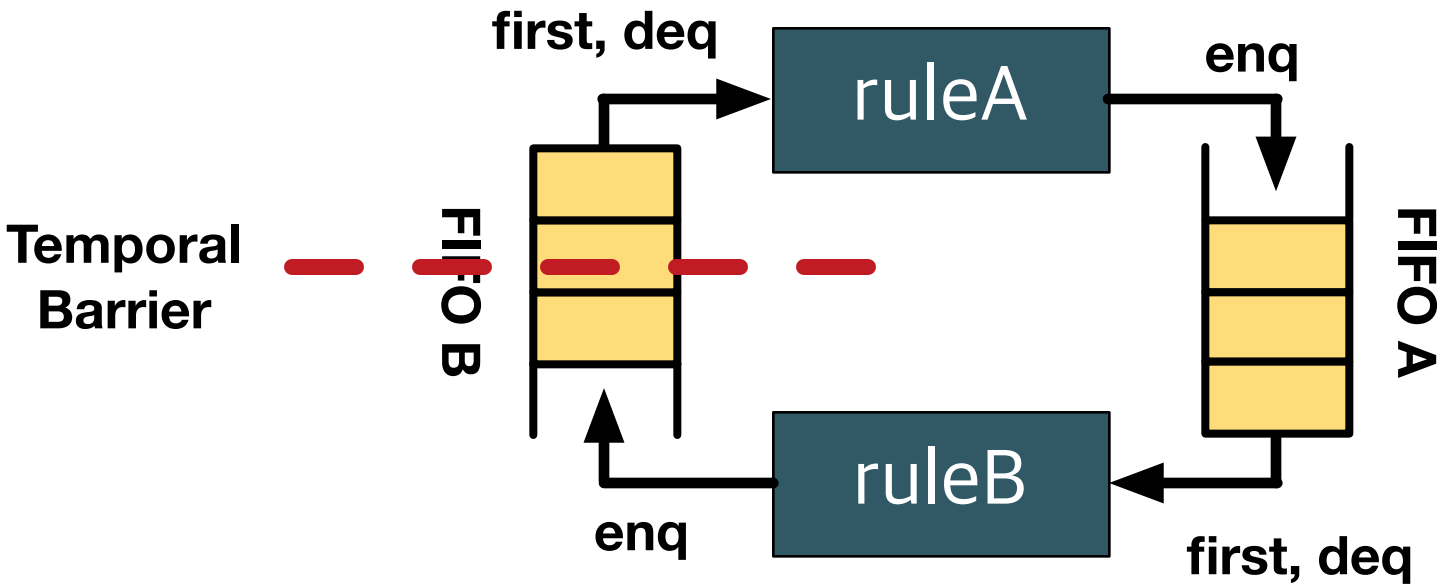


Because enqued data to a bypassFIFO canbe dequed at the same cycle, ruleA and ruleB forms a data dependence cycle

Solution?

Modules with Multiple Rules

- **Cyclic dependence**



We can delay the visibility of enqued data at a certain point.

This breaks the data dependence cycle within the same cycle

Modules with Multiple Rules

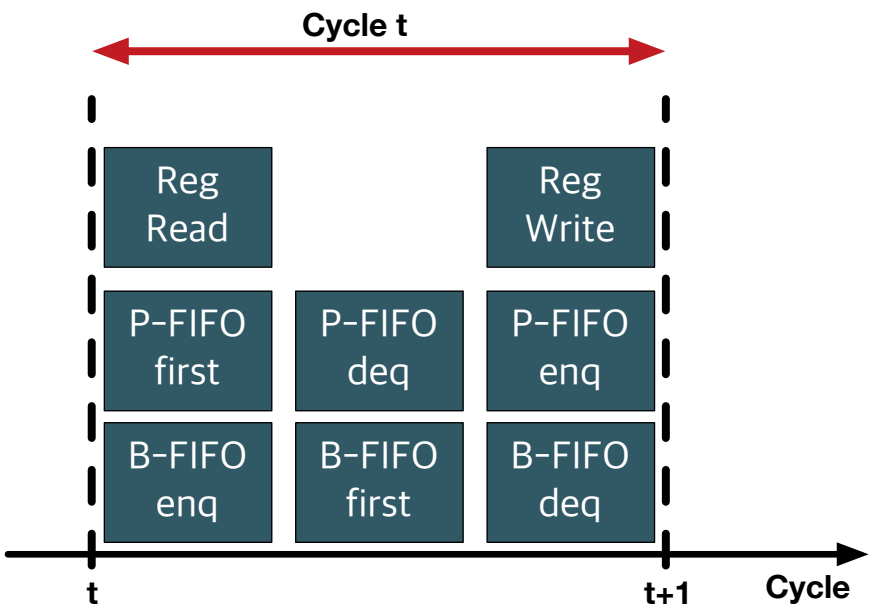
- **Cyclic dependence**

```
Fifo#(2, Bit#(16)) fifoA <- mkBypassFifo;
Fifo#(2, Bit#(16)) fifoB <- mkPipelineFifo;

rule ruleA;
    let data = fifoB.first; fifoB.deq;
    fifoA.enq(data-1);
    outputFifo.enq(data-1);
endrule

rule ruleB;
    let data = fifoA.first; fifoA.deq;
    fifoB.enq(data+1);
endrule
```

How to analyze the timing?

Method Scheduling Order

Module	Method scheduling order
PipelineFIFO	first < deq < enq
BypassFifo	enq < first < deq
Registers	read < write

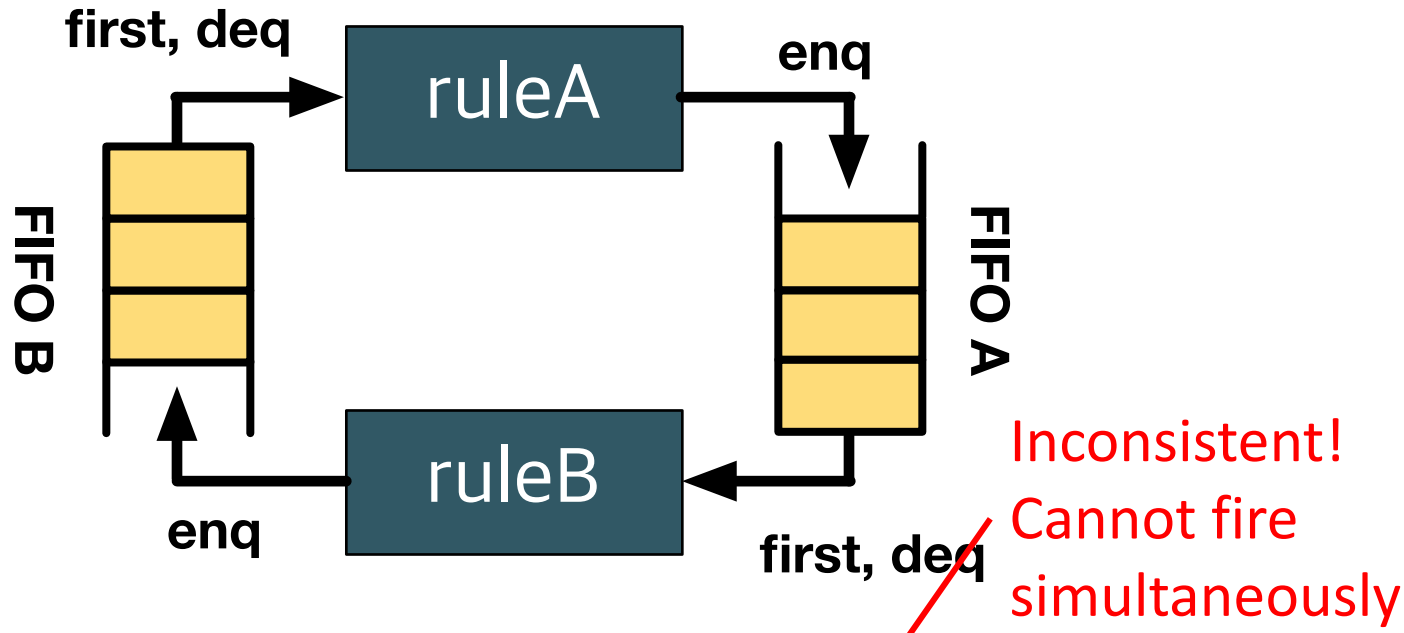


Order among methods of different modules is flexible

(e.g., P-FIFO first can be either before or after B-FIFO enq)

Rule Scheduling Analysis

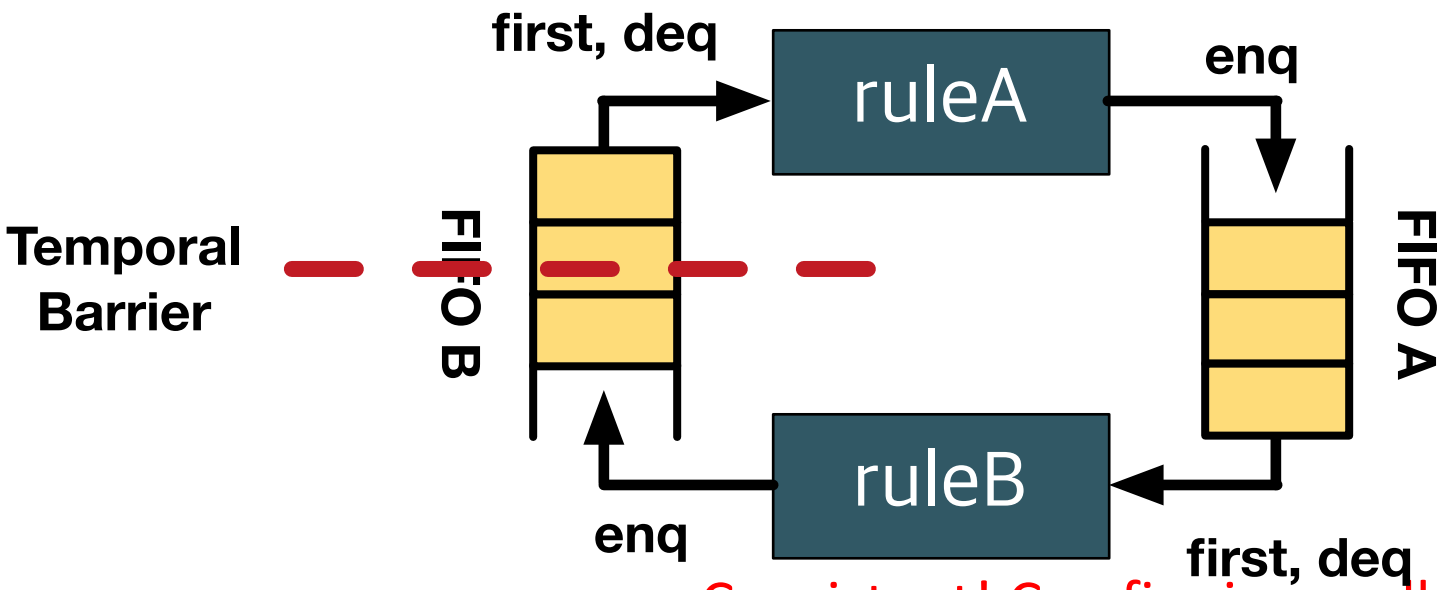
- **Original Version**



Submodules	ruleA	Order	ruleB
FIFOA	enq	<	deq, first
FIFOB	deq, first	>	enq

Rule Scheduling Analysis

- **Fixed Version**



Consistent! Can fire in parallel

Submodules	ruleA	Order	ruleB
FIFOA	enq	<	deq, first
FIFOB	deq, first	<	enq

Rule Guard

- **Revisiting fixed cyclic dependence example**

```
Fifo#(2, Bit#(16)) fifoA <- mkBypassFifo;
Fifo#(2, Bit#(16)) fifoB <- mkPipelineFifo;

rule ruleA (fifoA.notFull && fifoB.notEmpty);
    let data = fifoB.first; fifoB.deq;
    fifoA.enq(data-1);
    outputFifo.enq(data-1);
endrule

rule ruleB (fifoA.notEmpty && fifoB.notFull);
    let data = fifoA.first; fifoA.deq;
    fifoB.enq(data+1);
endrule
```

Implicit rule guard

(Submodule method availability in the statements of a rule becomes implicit rule guard)

A rule can fire only if its rule guard is true

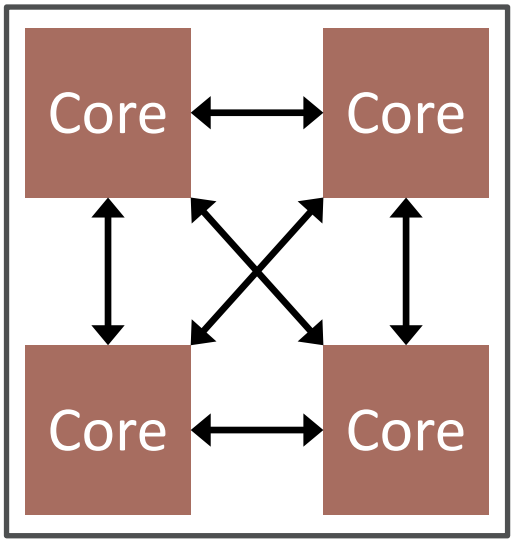
Day 2 Agenda

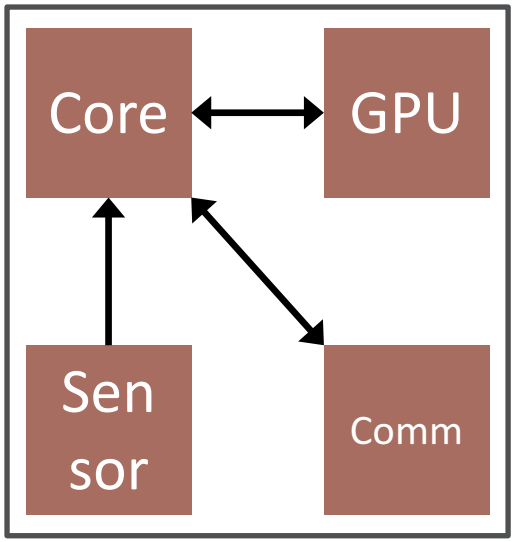
- **BSV Sequential Logic implementation and execution model**
  - Memory Elements
  - Latency-Inter-module Communication
  - Modules with Multiple Rules

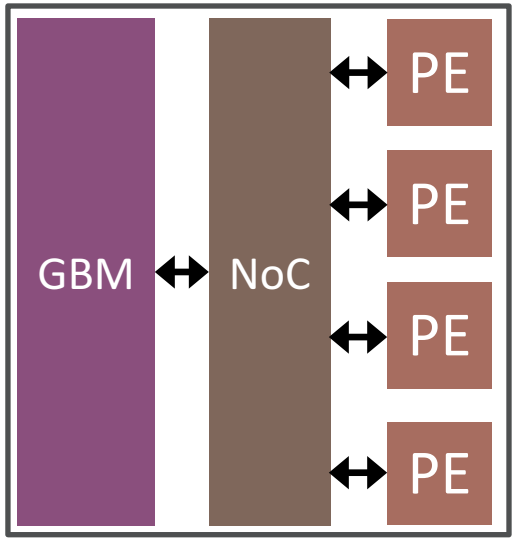
- **Traffic Patterns in CNN Accelerators**
  - Scatter
  - Gather
  - Local

- **Fixed Point Adder/Multiplier**

Traffic Patterns in Computer Systems







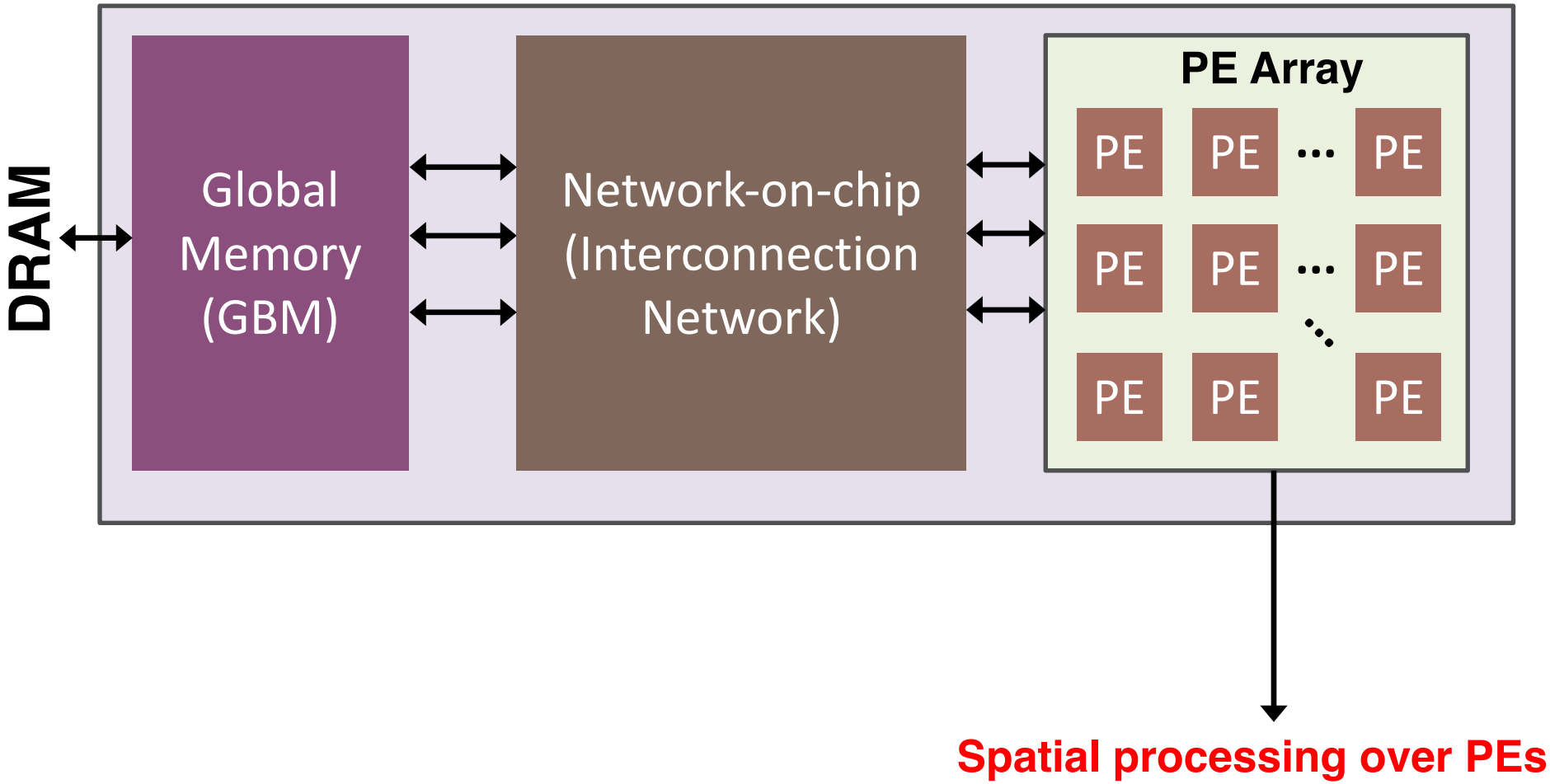
- **CMPs**
- **MPSoCs**
- **DNN Accelerators**

Dynamic all-to-all traffic

Static fixed traffic

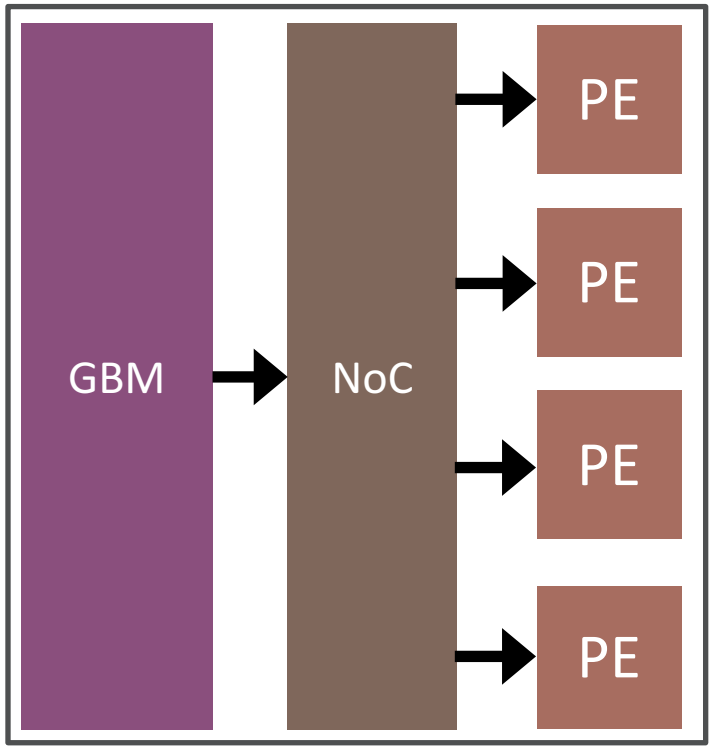
?

Spatial CNN Accelerator Structure

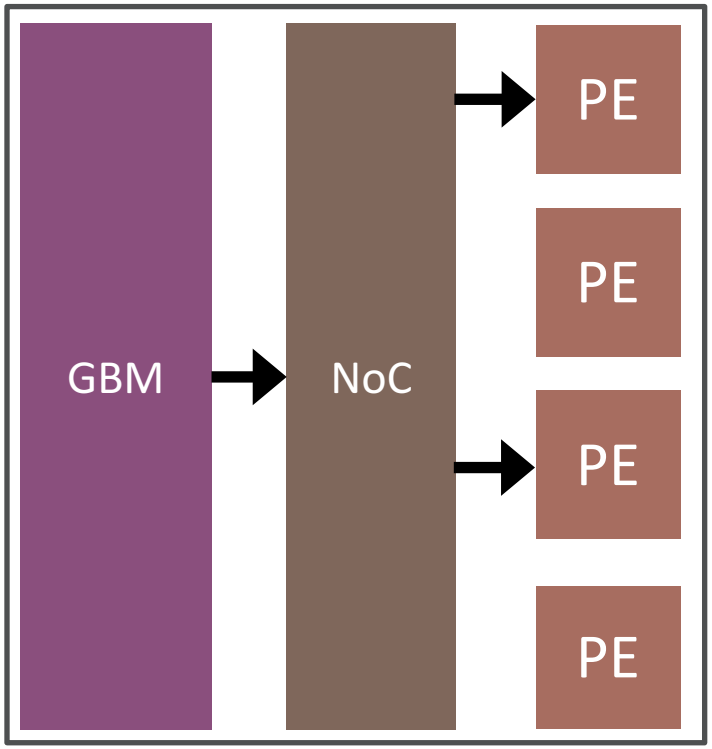


Traffic Patterns in CNN Accelerators

- **Scatter**



One-to-All

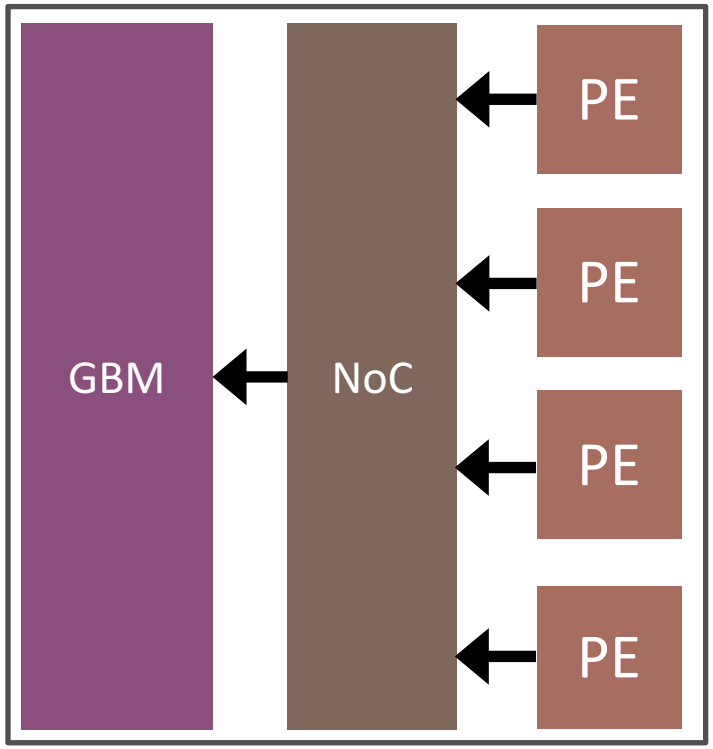


One-to-Many

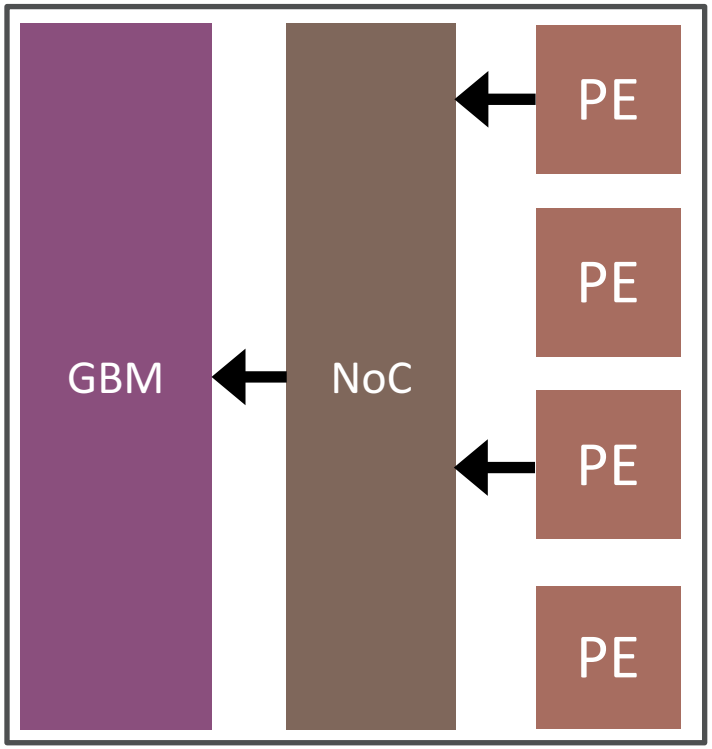
E.g., filter weight and/or input feature map distribution

Traffic Patterns in CNN Accelerators

- **Gather**



All-to-one

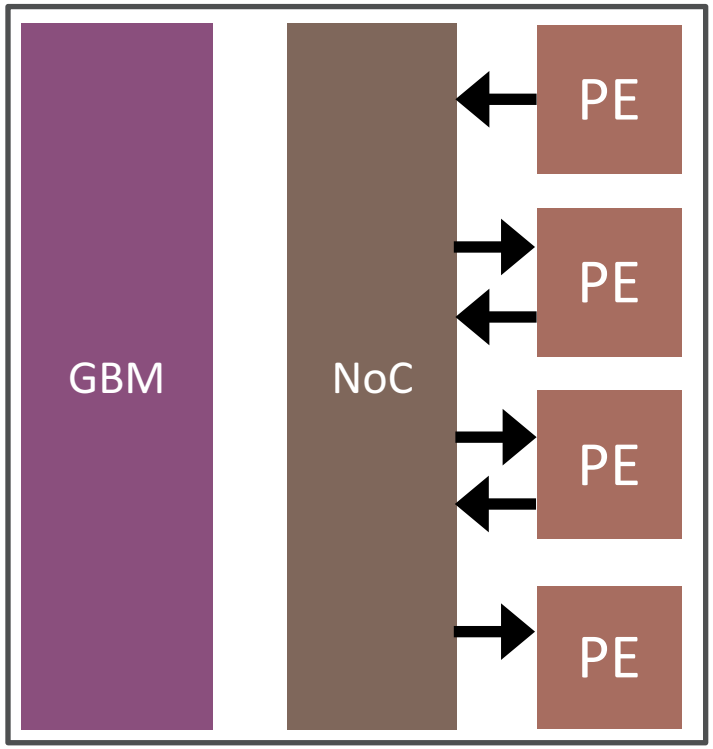


Many-to-one

E.g., partial sum gathering

Traffic Patterns in CNN Accelerators

- **Local**

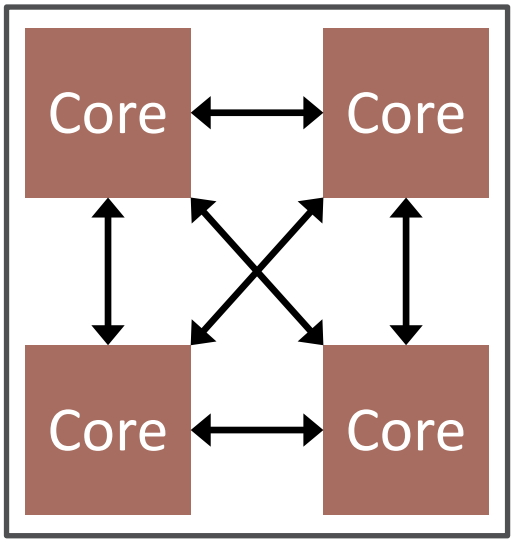


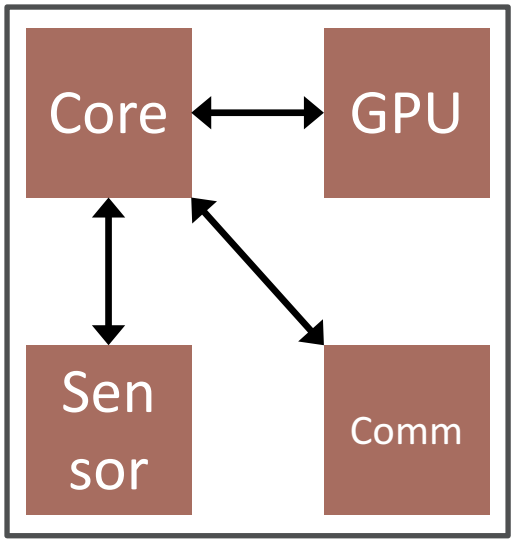
Many one-to-one

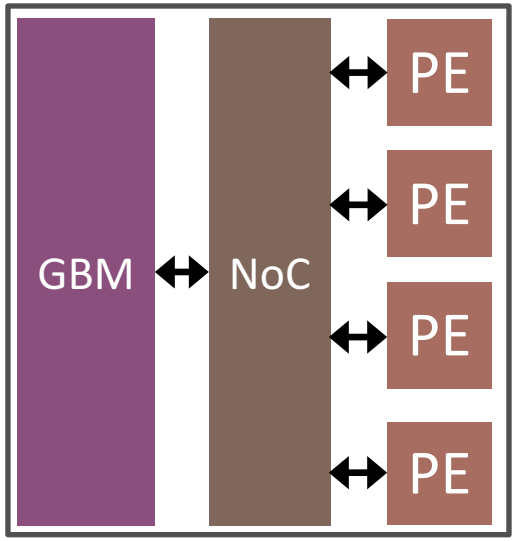
e.g., psum accumulation

- **Key optimization** to remove traffic between GBM and PE array and **maximize data reuse** in the PE array

Traffic Patterns in Computer Systems







- **CMPs**
- **MPSoCs**
- **DNN Accelerators**

Dynamic all-to-all traffic

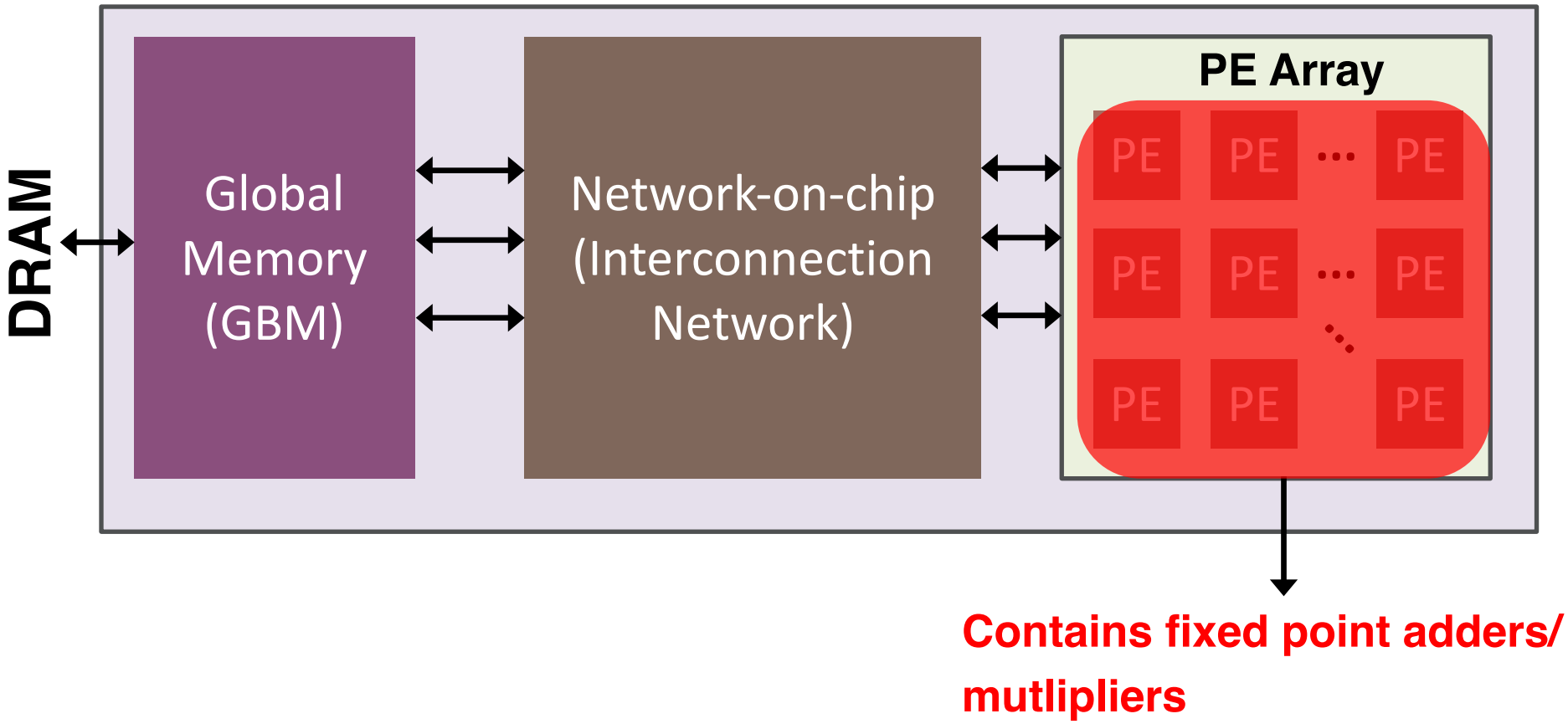
Static fixed traffic

Scatter
Gather
Local

Day 2 Agenda

- **BSV Sequential Logic implementation and execution model**
  - Memory Elements
  - Latency-Inter-module Communication
  - Modules with Multiple Rules
- **Traffic Patterns in CNN Accelerators**
  - Scatter
  - Gather
  - Local
- **Fixed Point Adder/Multiplier**

Spatial CNN Accelerator Structure



Contains fixed point adders/ mutlipliers

Fixed Point Arithmetic

- **Unsigned Fixed Point Representation**
  - Qn.m format: n-bit for integer bits m-bit for fractional bits (e.g., Q3.5 : 3-bit for integers and 5-bit for fractions.)
  - Example) 010.10100 = 2 + ½ + 1/3 = 2.625

2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
0	1	0		1	0	1	0	0

Fixed Point Arithmetic

- **Signed Fixed Point Representation**
  - Represent in 2's complement format
  - Recall that the MSB (sign-bit) in a signed binary number actually represents $-2^{(m-1)}$, where m is the number of bits in a binary number. (e.g., $1011_2 = -2^3 + 2^1 + 2^0 = -5$)
  - Example) -3.25 = -4 + 0.75 = 100.0000 + 000.1100 = 100.1100

-2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
1	0	0		1	1	0	0	0

Fixed Point Arithmetic

- **Signed Fixed Point Addition**
  - The same process as binary integer addition
  - Example) -3.25 + 2.625 = 100.11000 + 010.10100 = 111.01100 = -4 + 3.375 = -0.625

-2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
1	0	0		1	1	0	0	0
0	1	0		1	0	1	0	0
+								
1	1	1		0	1	1	0	0

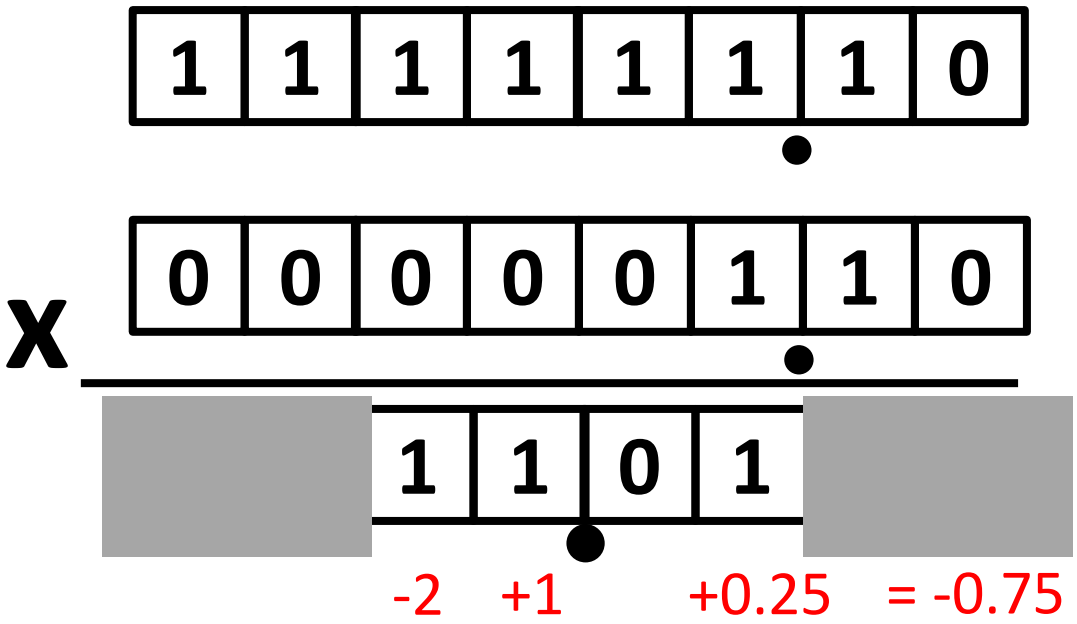
Fixed Point Arithmetic

- **Signed Fixed Point Multiplication**
  - The same process as binary integer multiplication
    1) Sign-extend each operand (double bit width of original)
    2) Perform binary integer multiplication
    3) Truncate extra bits for integer and fraction bits independently

Fixed Point Arithmetic

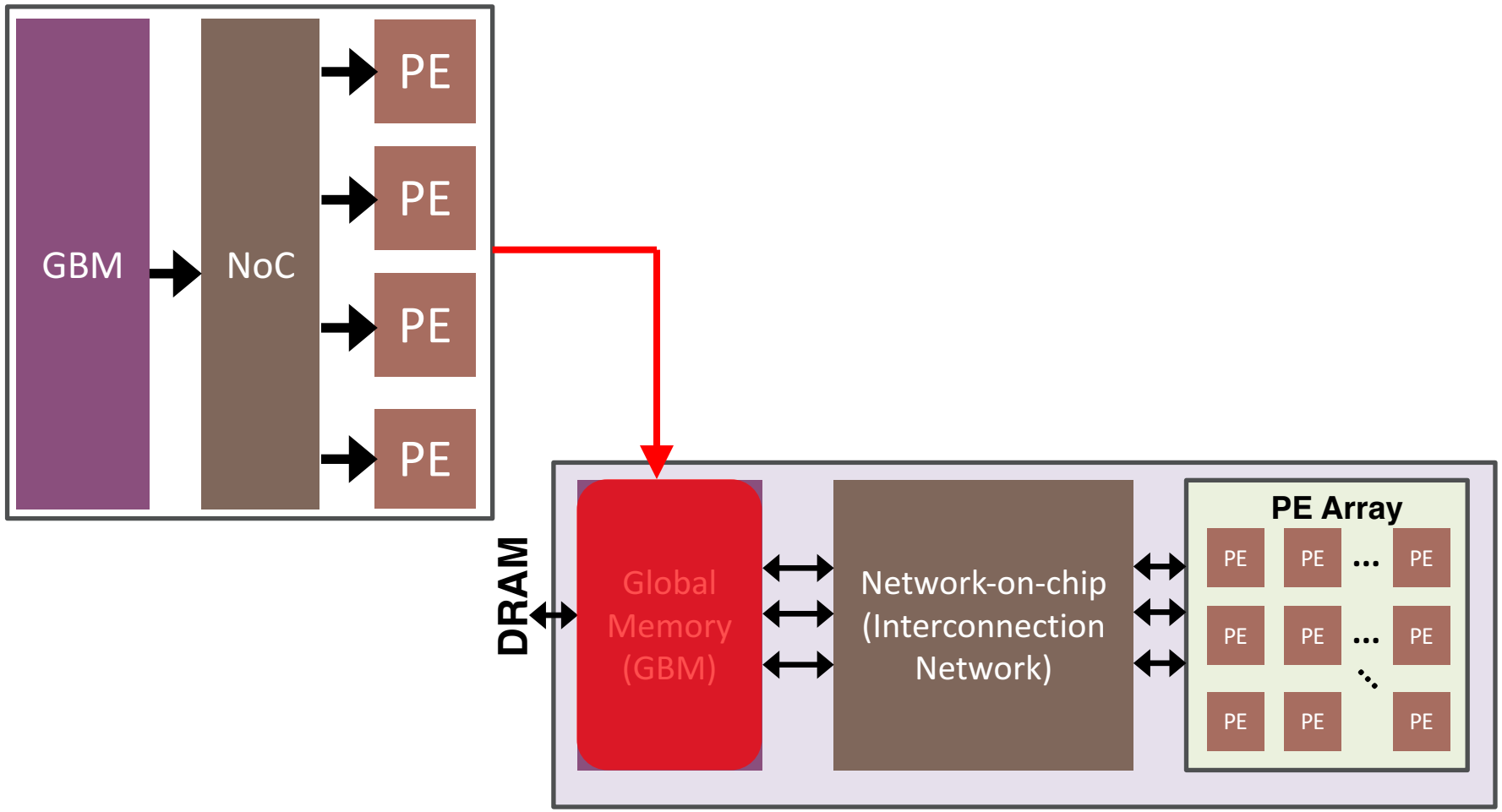
- **Signed Fixed Point Multiplication**
  - **Example**) Using Q1.2 format;

    - 0.5 x 1.5 = -0.75



[Lab1] DataReplicator

- **Repeating Data to Support Broadcasting**



[Lab1] Data Replicator

- **Module Description**
  - External module requests data repeat using “putData” method

    method Action putData(RepData value, RepIdx numRepeats)

  - Another external module receives data using “getData” method

    method ActionValue#(RepData) getData

- **Spec**
  - DataReplicator module repeats putting “value” for “numRepeats” times to the method getData

[Lab1] Data Replicator

- **Example**

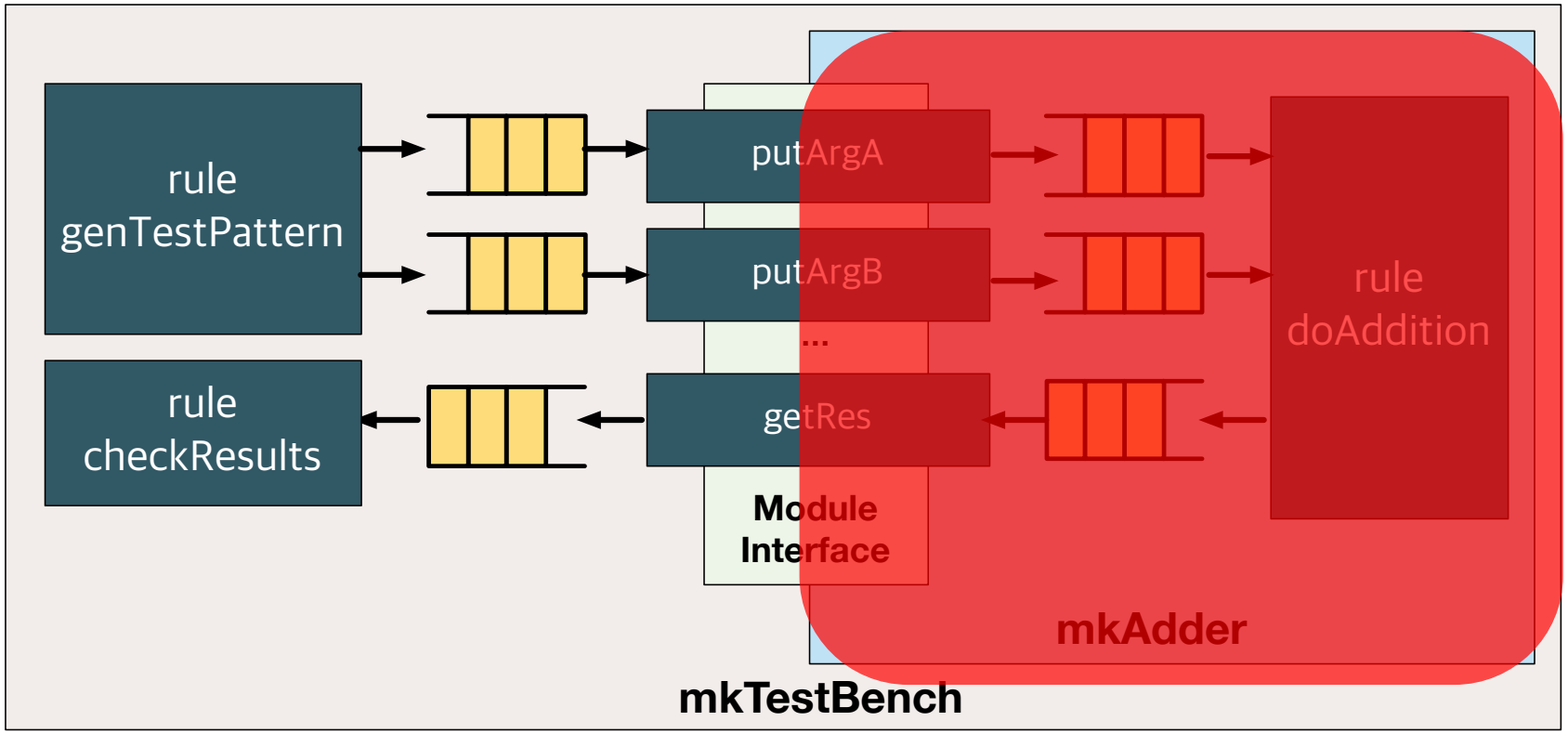
```
rule genTestPattern;
  replicator. putData(15, 3); // Repeat 15 three times
endrule

rule checkOutput;
  let outData <- replicator.getData;
  $display("Received %d", outData);
endrule
```

- **Print-out message**

```
Received 15
Received 15
Received 15
```

[Lab2] Fixed Point Adder and Multiplier

- **Designing fixed point adder / multiplier**



TODO

[Lab2] Fixed Point Adder and Multiplier

- **Spec**
  - Fixed point type: Q3.12 (sign-bit + 3 integer bits + 12 fraction bits = 16 bit)
  - For module interface, implement LI interface
    - All the input/output FIFOs are pipelineFIFO
  - Addition / multiplication takes one cycle
  - Use “+” and “ * ” to perform binary integer addition / multiplication (don’t need to implement your own adder/multiplier)

- **Useful statements**
  - Bit extension: signExtend() / zeroExtend()
  - Bit selection: [] (e.g., **Bit**#(6) a = 6’b11010010;
    // a[7:5] == 3’b110
    // a[0] = 1’b0)

[Lab2] Fixed Point Adder and Multiplier

- **Advanced topic [optional]**
  - Parameterize the adder / multiplier so that your adder/multiplier works with any fixed point settings

- **Useful statement examples (hints)**
  - **typedef** 5 IntegerBits;
  - **typedef** TAdd#(IntegerBits, TAdd#(SignBits, FractionBits) FixedBits;
  - **Bit**#(IntegerBits) intBits;
  - intBits = fixedBits [valueOf(FixedBits) – valueOf(SignBits) -1 : valueOf(fractionBits)];
  - **Bit**#(TAdd#(FixedBits, FixedBits)) extendedBit;