CS 250B: Modern Computer Systems

FPGA Accelerator Design Principles
Part 1: Designing Pipelines

Sang-Woo Jun

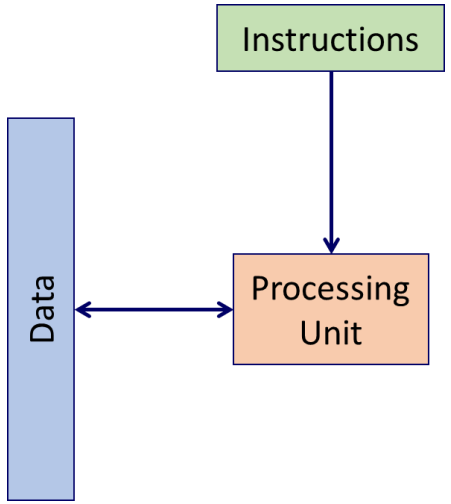
UCI

System Design For High Performance

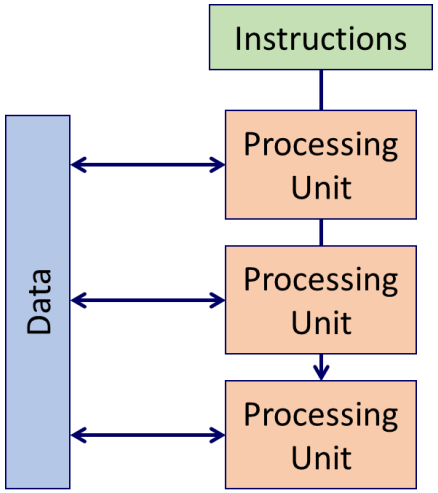
- Designing hardware for low delay -> High clock speed
- Designing hardware for high throughput

- Techniques including
  - Pipelining
  - Wider datapath width
  - Latency and duty cycle
  - Implementing computation as filters

Pipelining: Back to Flynn's Taxonomy

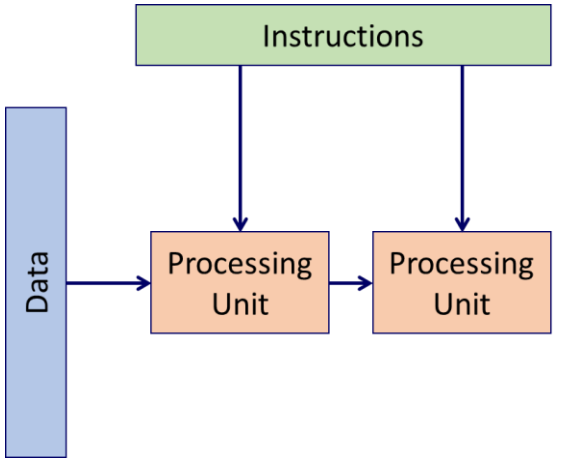


SISD
Single-Instruction
Single-Data
(Single-Core Processors)



SIMD
Single-Instruction
Multi-Data
(GPUs, SIMD Extensions)

CPU/GPU



MISD
Multi-Instruction
Single-Data
(Systolic Arrays,…)

FPGA target

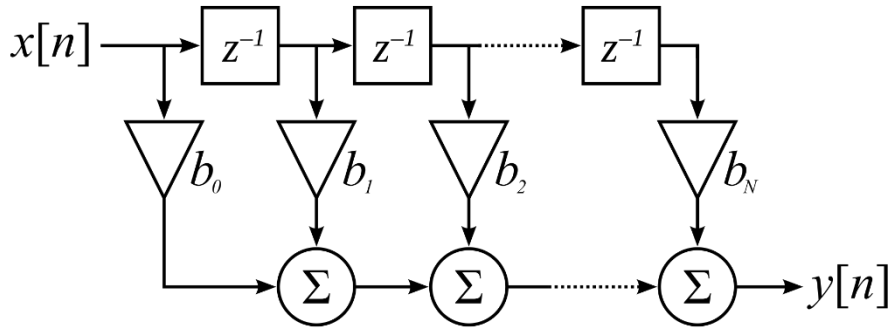
Implementing Computation as Filters

- Filter: Computation unit looking at a small window in a data stream, emits a different stream
  - Colloquial definition…
  - Can be implemented architecturally, via MISD systolic array
- Only looks at a small window -> Small memory requirement!
  - Even if input/output data is large, no need to store it all anywhere
  - Everything is processed in a streaming manner
- Reduces memory requirement
  - On-chip BRAM resources are precious
  - If we can avoid going to off-chip memory, great performance save!

Example:
Finite Impulse Response (FIR) Filter

- "Discrete signal processing filter whose response to a signal is of finite length"
- Simple intuition: Each output of the filter is a weighted sum of N most recent input values
  - Convolution of a 1D matrix on a streaming input
  - Used to implement many filters including low-pass, band-pass, high-pass, etc
  - Weights are calculated using Discrete Fourier Transforms, etc

$$y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i]$$

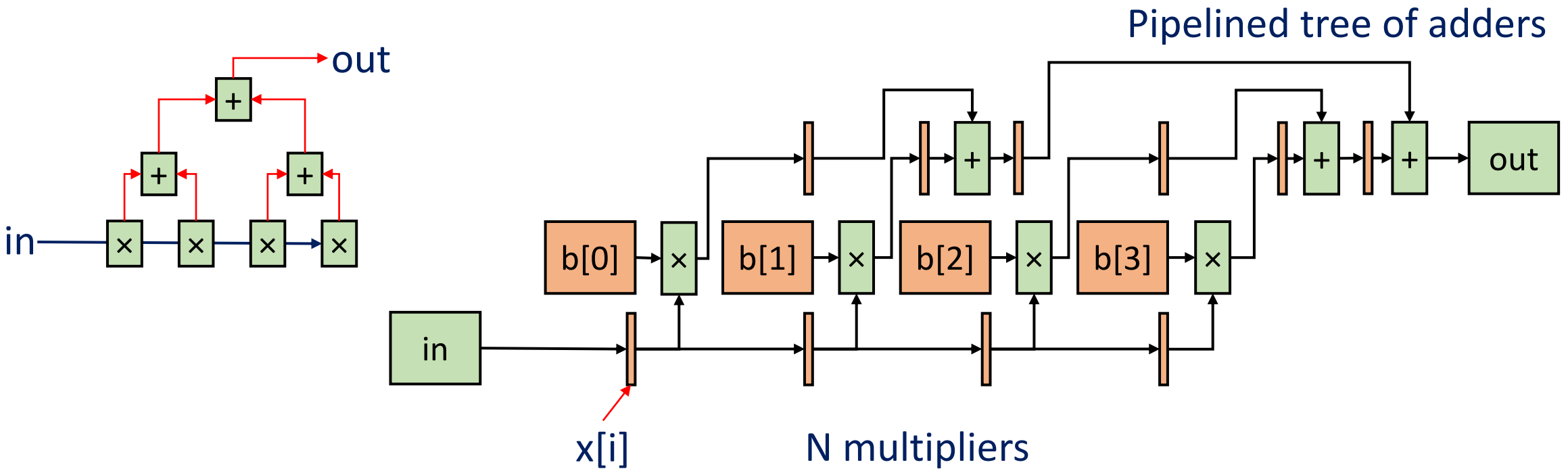


Naïve, Non-Filter Solution

- Store all input in array, and loop over it
  - Two layer FOR loops, one over array, one over window (i-N to i)
  - Either store or emit results of each 1st-level loop
  - Uses one ALU, CPU-style sequential programming approach (1 op per cycle)
- This solution is very inefficient in hardware
  - First, needs memory to store array
    - Either precious on-chip memory, or expensive off-chip memory
  - Second, only one element processed each cycle
    - One cycle ~= One inner loop iteration -> N cycles per ouput!
    - Better if we can break array into parallel portions, but adds circuit space overhead for managing computational state of each processing element

Pipelining an Finite Impulse Response (FIR) Filter

- Every pipeline stage multiplies x[i]*b[j] and forwards accumulated result
  - For window size of N, instantiate N multipliers!
  - N multiplications per cycle + log(N) adders per cycle
  - For N == 4, 7 ops per cycle!



FIR Filter In Bluespec

```
 1 import FIFO::*;
 2 import Vector::*;
 3
 4 interface FirFilterIfc;
 5 →   method Action enq(Bit#(32) data);
 6 →   method Action deq;
 7 →   method Bit#(32) first;
 8 endinterface
 9
10
11 typedef 2 FilterWindow;
12 module mkFirFilter(FirFilterIfc);
13 →   Vector#(FilterWindow, Bit#(32)) weights;
14 →   weights[0] = 5;
15 →   weights[1] = 8;
16 →   Vector#(FilterWindow, FIFO#(Bit#(32)) internalQ <- replicateM(mkFIFO);
17 →   Vector#(FilterWindow, FIFO#(Bit#(32)) addQ <- replicateM(mkFIFO);
18 →   FIFO#(Bit#(32)) outputQ <- mkFIFO;
```

```
38 →   method Action enq(Bit#(32) data);
39 →   →   internalQ[0].enq(data);
40 →   endmethod
41 →   method Action deq;
42 →   →   outputQ.deq;
43 →   endmethod
44 →   method Bit#(32) first;
45 →   →   return outputQ.first;
46 →   endmethod
47 endmodule
```

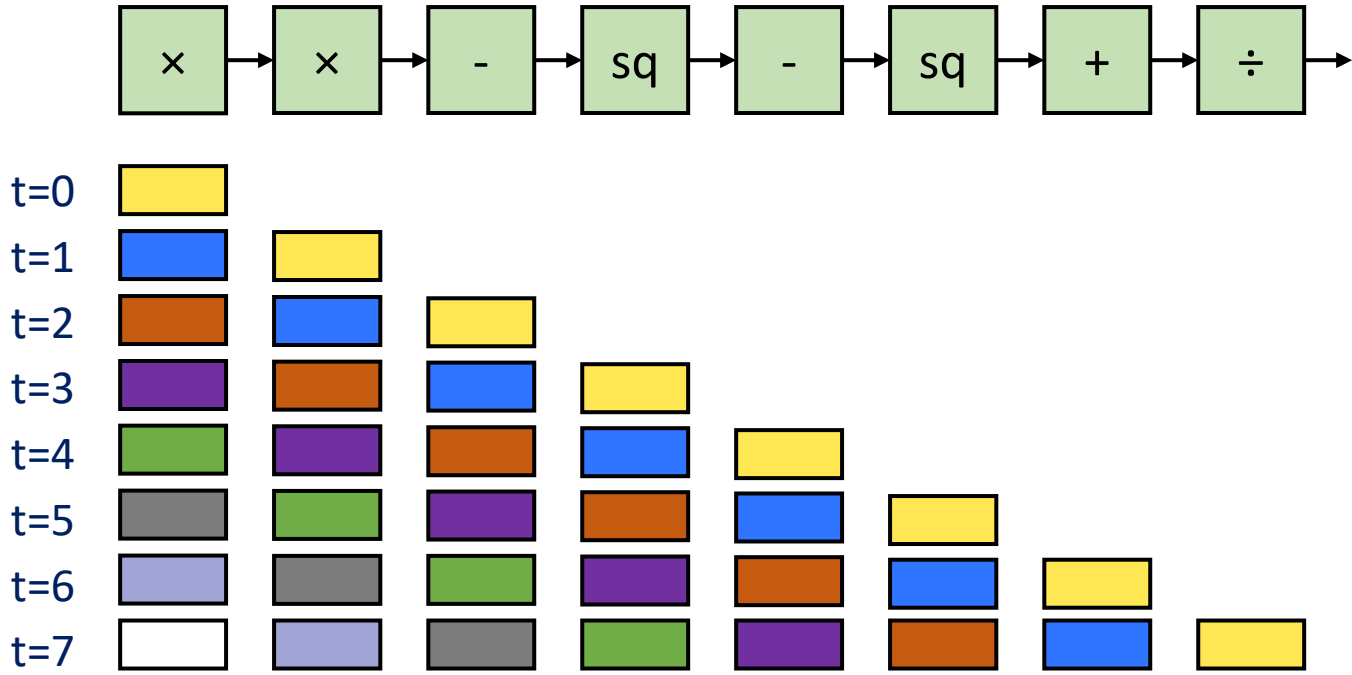
Static Elaboration!

```
17 →   for ( Integer i = 0; i < valueOf(FilterWindow); i=i+1 ) begin
18 →   →   rule calcWeight;
19 →   →   →   internalQ[i].deq;
20 →   →   →   let d = internalQ[i].first;
21 →   →   →   addQ[i].enq( d * weights[i] );
22 →   →   →   if ( i < valueOf(FilterWindow)-1 ) internalQ[i+1].enq(d);
23 →   →   endrule
24 →   end
25
26 →   ...
```

```
20 →   rule calcWeight1;
21 →   →   internalQ[0].deq;
22 →   →   let d = internalQ[0].first;
23 →   →   addQ[0].enq( d * weights[0] );
24 →   →   internalQ[1].enq(d);
25 →   endrule
26 →   rule calcWeight2;
27 →   →   internalQ[1].deq;
28 →   →   addQ[1].enq( internalQ[1].first * weights[1] );
29 →   endrule
30
31 →   rule calcAdd;
32 →   →   addQ[0].deq;
33 →   →   addQ[1].deq;
34 →   →   outputQ.enq(addQ[0].first + addQ[1].first);
35 →   endrule
```

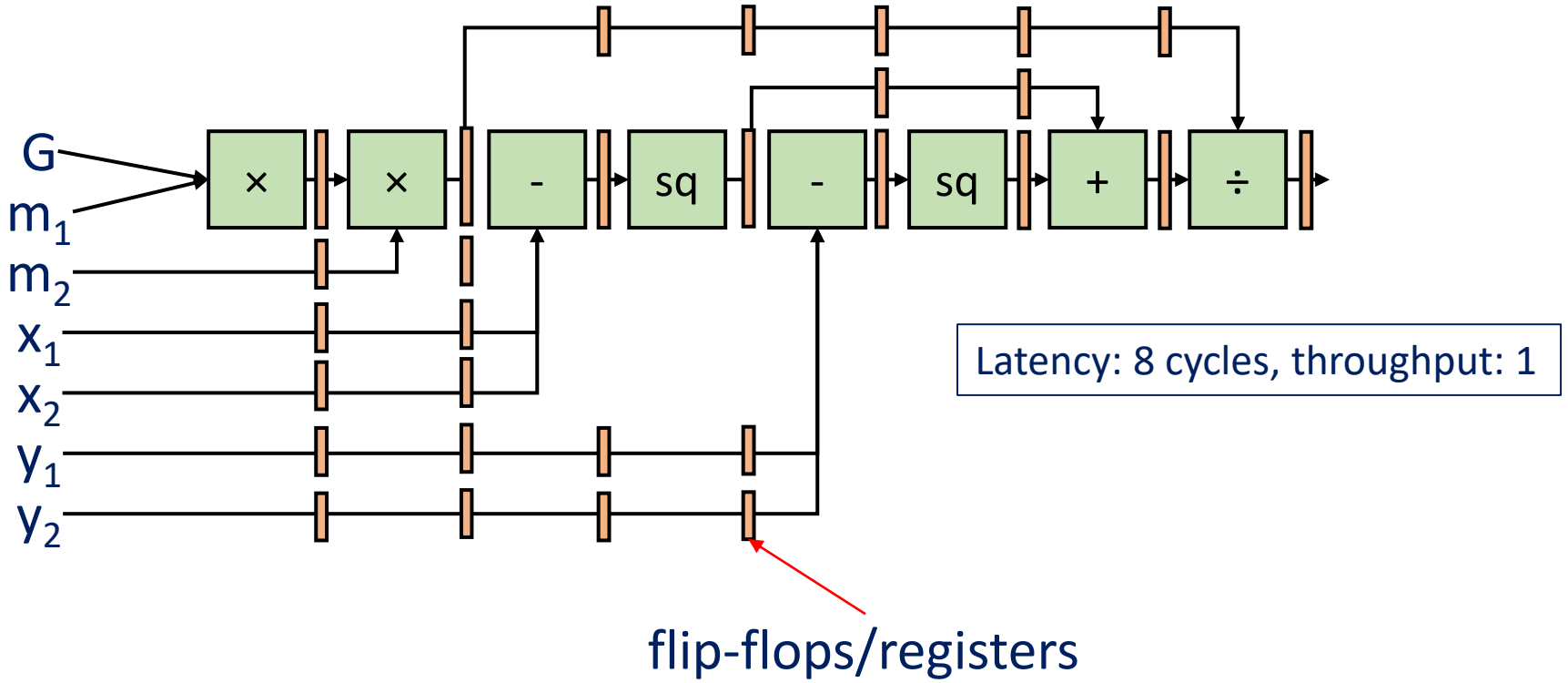
Example Application: Gravitational Force

- $\frac{G \times m_1 \times m_2}{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
- 8 instructions on a CPU – 8 cycles* Complicated with superscalar, OoO
  - One answer per 8 cycles
  - Ignoring load/store
- FPGA: Pipelined systolic array
  - One answer per 1 cycle
  - 200 MHz vs. 3 GHz makes a bit more sense!
  - Very simple systolic array
    - We can fit a lot into a chip
    - Unlike cache-coherent cores



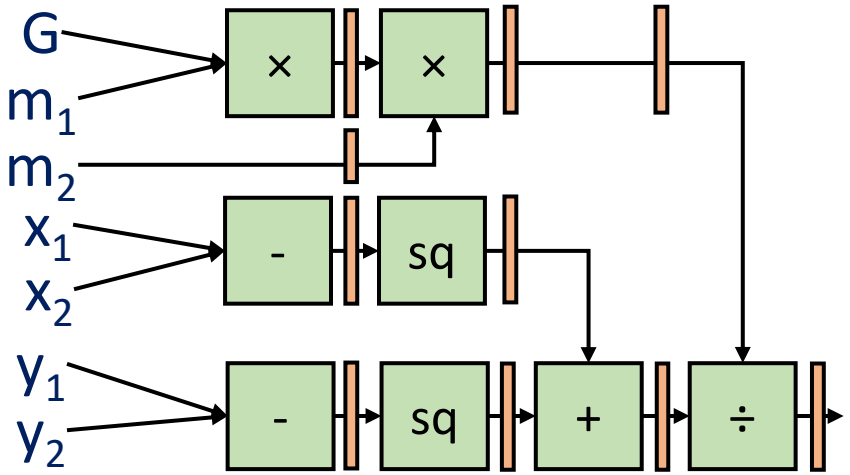
Pipelined Implementation Details

$$\frac{G \times m_1 \times m_2}{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$



Pipelined Implementation Details

$$\frac{G \times m_1 \times m_2}{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$



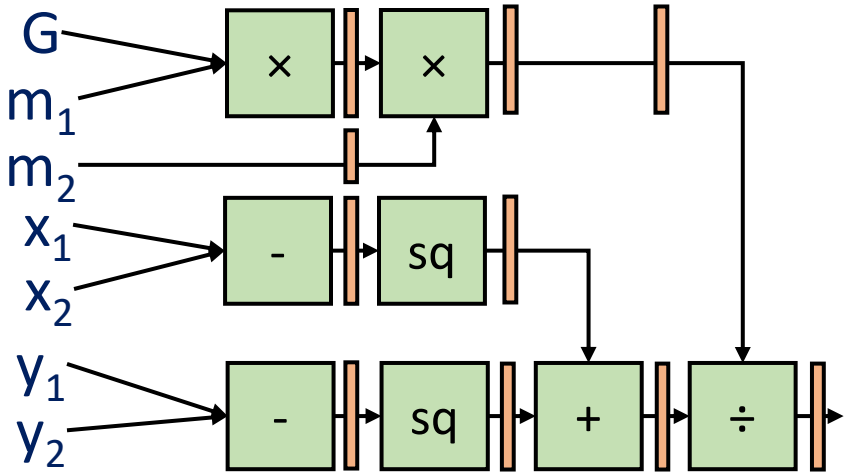
Latency: 4 cycles, throughput: 1

Latency doesn't matter too much if this is simple stream processing.
If later computation depends on earlier results, latency becomes important!

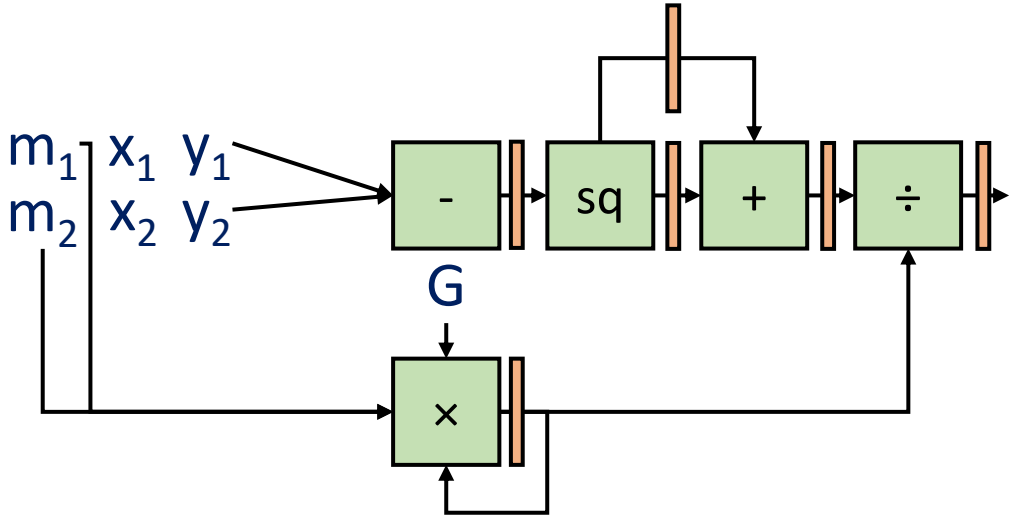
6 input elements per cycle (excluding G)! 24 bytes! Does our bus support it?

Pipelined Implementation Details

$$\frac{G \times m_1 \times m_2}{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$



Latency: 4 cycles, throughput: 1



Latency: 5 cycles, throughput: 1/3

Important Aside: Little's Law

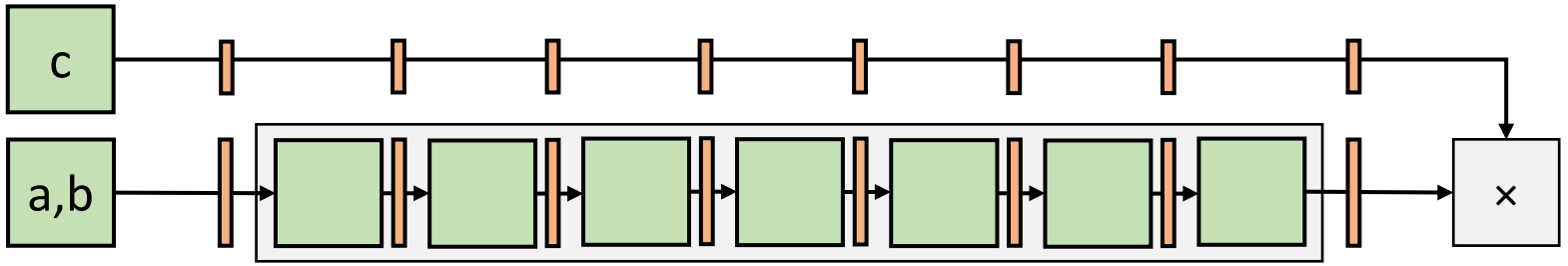
- $L = \lambda W$
  - L: Number of requests in the system
  - λ: Throughput
  - W: Latency
- Simple analysis: Given a system with latency W, in order to achieve throughput, we need to be processing at least L things at once

Example: Floating Point Arithmetic in FPGA

- Floating point operations implemented using CLBs is inefficient
  - CLB: Configurable Logic Blocks – Basic unit of reconfiguration
- Xilinx provides configurable floating point “cores”
  - Uses DSP blocks to implement floating point operations
  - Variable cycles of latency – variable maximum clock speed
  - Configurable operations (+-*/…) and bit width (single/double/other)
  - e.g., Single-precision (32-bits) floating point multiplication can have 7 cycles of latency for the fastest clock speed/throughput

Example: Floating Point Arithmetic in FPGA

- Simple example: a×b×c
  - Uses two multiplier cores, 7 cycle latencies each
  - c also needs to be forwarded via 7 pipelined stages
- What happens if c has less than 7 forwarding stages?
  - From $L = \lambda W$: If we have L = 4, W = still 7 → λ=4/7



If a,b,c are not inserted every cycle, some cycles may emit garbage answers
Each intermediate value must be tagged with valid/invalid tag! Or…

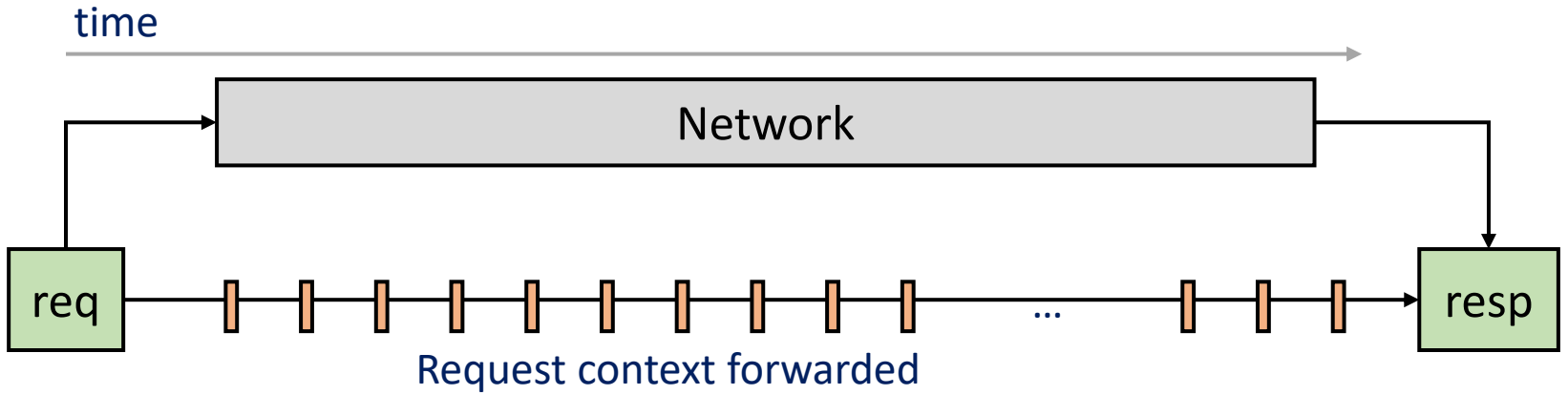
Floating Point Arithmetic in Bluespec

- Forwarding stages abstracted using a FIFO
  - FIFO length doesn't have to exactly match latency – Only has to be larger
    - Some freedom of internal implementation
  - Latency-insensitive (or less sensitive design)
- If fifo.notEmpty, valid result!

```
FloatMult#(8,23) mult1 <- mkFloatMult;
FloatMult#(8,23) mult2 <- mkFloatMult;
FIFO#(Float) cQ <- mkSizedFIFO(7);
rule step1;
  Tuple3#(Float,Float,Float) a = inputQ.first;
  inputQ.deq;
  cQ.enq(tpl_3(a));
  mult1.put(tpl_1(a),tpl_2(a));
endrule
rule step2;
  Float r <- mult1.get;
  cQ.deq;
  mult2.put(r, cQ.first);
endrule
rule step3;
  let r <- mult2.get;
  outputQ.enq(r);
endrule
```

Another example: Memory/Network Access Latency

- Round-trip latency of DRAM/Network on FPGA is in the order of microseconds
- Let's assume 2 us network round-trip latency, FPGA at 200 MHz
  - Network latency is 100 cycles
  - Little's law means 100 requests must be in flight to maintain full bandwidth, meaning the forwarding pipeline must have more than 100 stages
  - mkBRAMFIFO



Replicating High-Latency Pipeline Stages

- Some pipeline stages may inherently have high latency, but cannot handle many requests in flight at once
  - The GCD module from before is a good example (Large W, but L=1)
  - Replicate the high-latency pipeline stage to increase L
- Example: In-memory sorting
  - Sorting 8-byte element in chunks of 8 KB (1024 elements)
  - Sorting module using binary merge sorter requires 10 passes to sort 8 KB
    - 1024*9 cycle latency (last pass can be streaming)
    - Little's law says we need 1024*9 elements in flight to maintain full bandwidth
    - Sorter module needs to be replicated 9 times to maintain wire-speed

Analysis of the Floating Point Example

- Computation: a*b*c
- Let's assume clock frequency of 200 MHz
- Input stream is three elements (a,b,c) per cycle = 12 Bytes/Cycle
  - 200 MHz * 12 Bytes = 2.4 GB/s
  - Fully pipelined implementation can consume data at 2.4 GB/s
  - If we're getting less than that, there is a bottleneck in computation!
- What if we want more than 1 tuple/cycle?

Increasing Datapath Width

- Increase the amount of data processed each cycle per pipeline
  - Performance loss if the datapath is slower than processing
  - If clock is fixed, datapath speed increased by widening
- Processing then may be modified for even higher throughput
  - Replication, algorithm redesign, pipelining, …

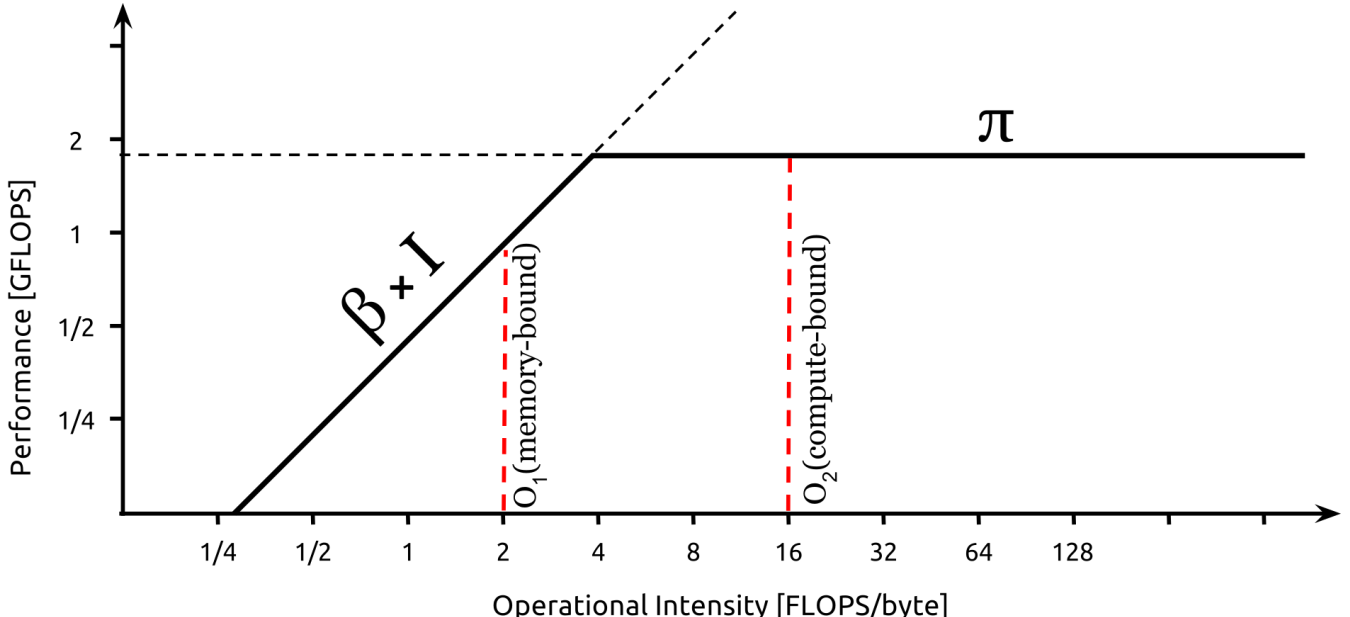
8 Bytes/Cycle @ 200 MHz = 1.6 GB/s

32 Bytes/Cycle @ 200 MHz = 6.4 GB/s

Important Aside: Roofline Model

- Simple performance model of a system with memory and processing
- Performance is bottlenecked by either processing or memory bandwidth
  - As compute intensity (computation per memory access) increases, bottleneck moves away from memory bandwidth to computation performance
  - Not only FPGAs, good way to estimate what kind of performance we should be getting in a given system

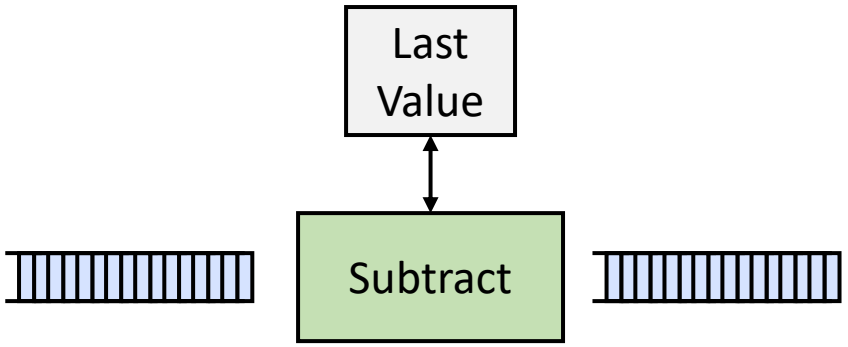
We don't want our computation implementation to be the bottleneck!
Goal: Saturate PCIe/Network/DRAM/etc



Example: Delta Compression

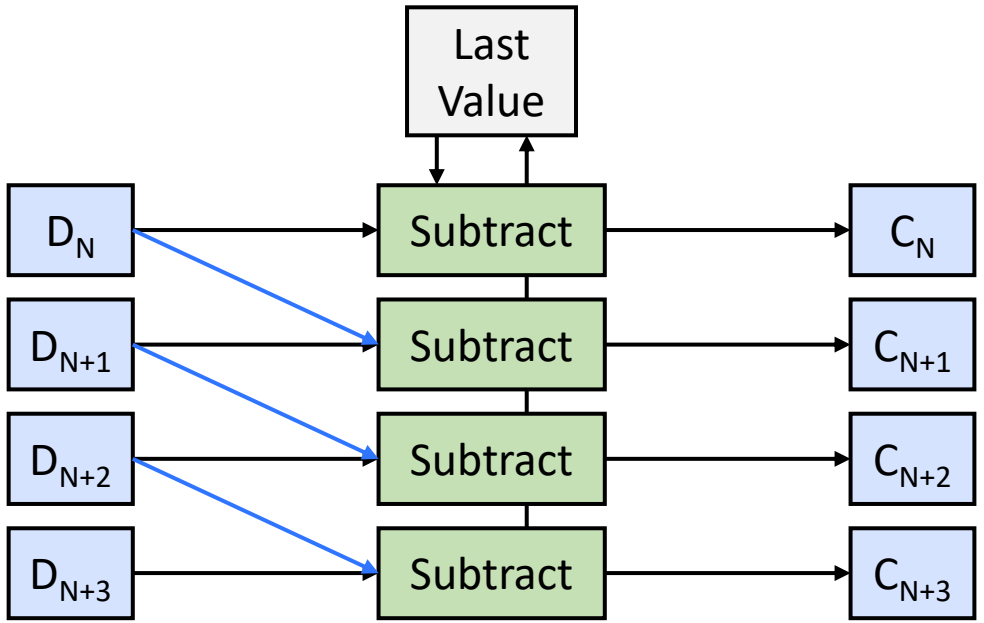
- For each element in array, subtract the previous element from the current one
  - If the delta is typically low, this reduces the average element size
  - Efficient encoding can use less bits for smaller values
- Naïve implementation keeps one “last” value, initialized to zero
  - Processes one element per cycle
  - For 32 bit elements, 4 bytes per clock cycle (@ 200 MHz, 800 MB/s)

NVMe SSDs provide ~4 GB/s!



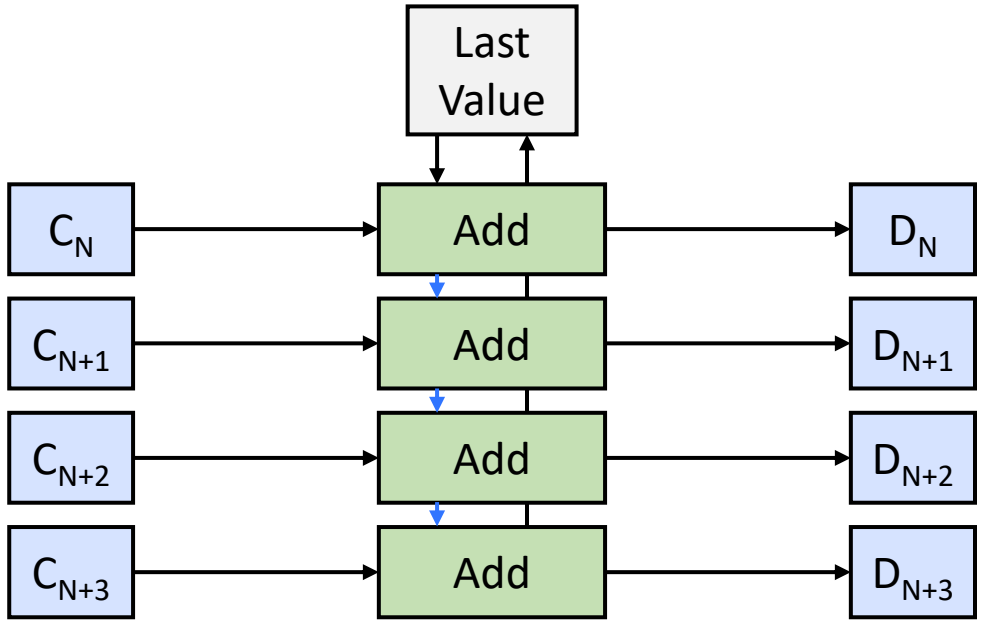
Example: Delta Compression

- One solution: replicate engines
  - Stream needs to be broken into independent chunks
  - If processing expects in-order processing, chunks must be reassembled in order
    - Requires memory for temporary sorted data
- Alternative: Wider datapath
  - Easy for encoding! Independent per element
  - For 32 bit elements, 16 bytes per clock cycle (@ 200 MHz, 3.2 GB/s)



Example: Delta Decompression

- Decoding is a bit more complicated
  - D_{N+1} now depends on decoded D_N
  - New “Last Value” depends on all D data
    - Very long combinational path, with a chain of 4 adders
- Most likely will reduce clock speed

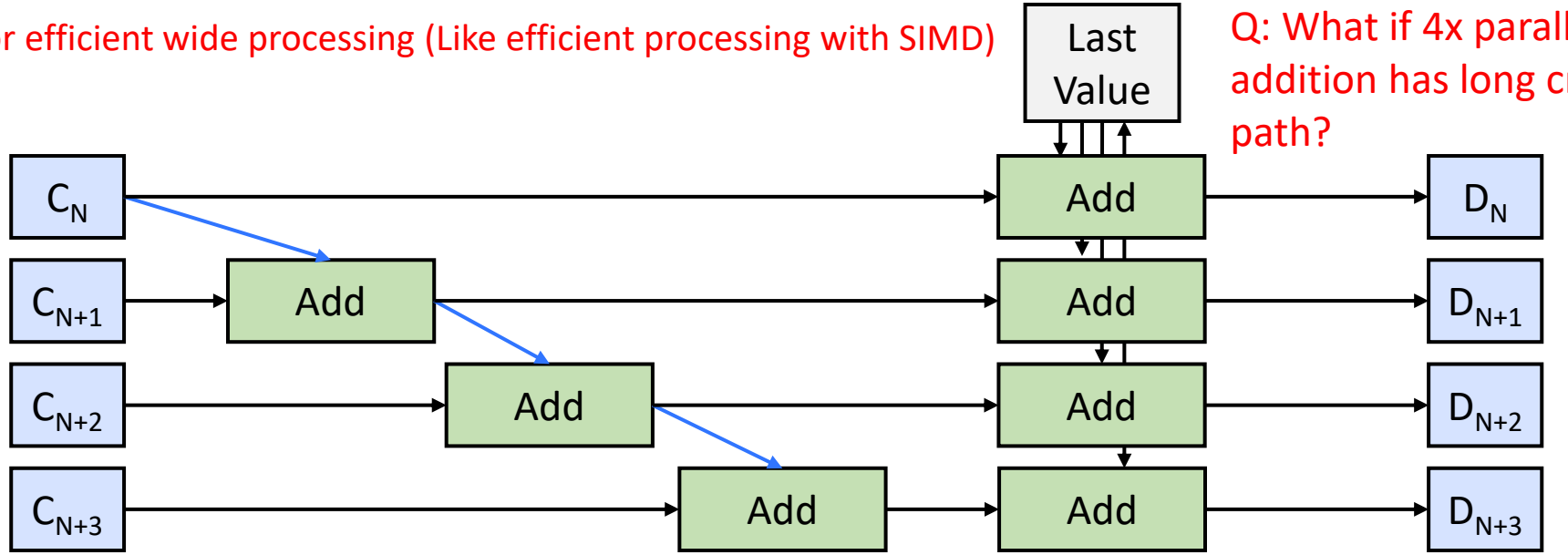


Example: Delta Decompression

- One solution: Pipeline!
  - The chained addition can be done first, in a pipelined way
    - No dependency across tuples yet, so latency not a big issue
  - Adding last value done at once after all chain adds are done
    - Update last value in same cycle for the next input tuple

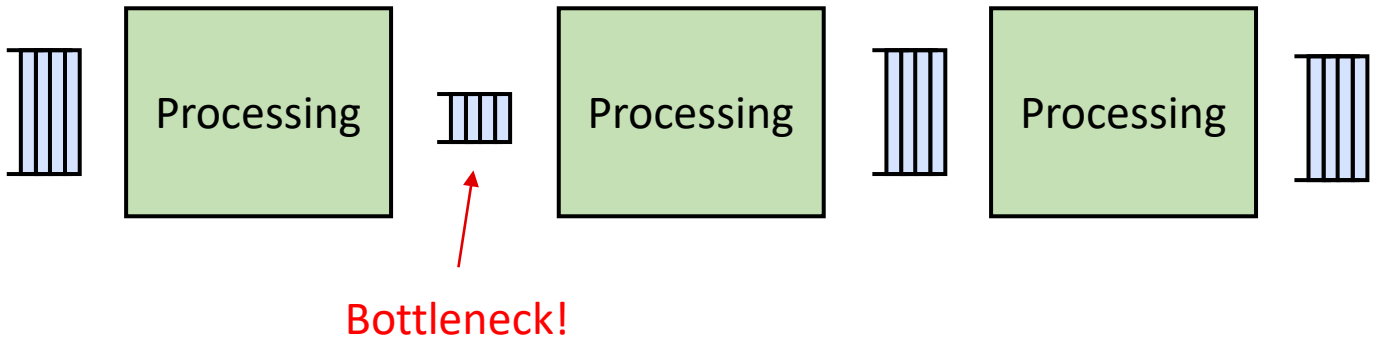
Some thinking is often necessary for efficient wide processing (Like efficient processing with SIMD)

Q: What if 4x parallel addition has long critical path?



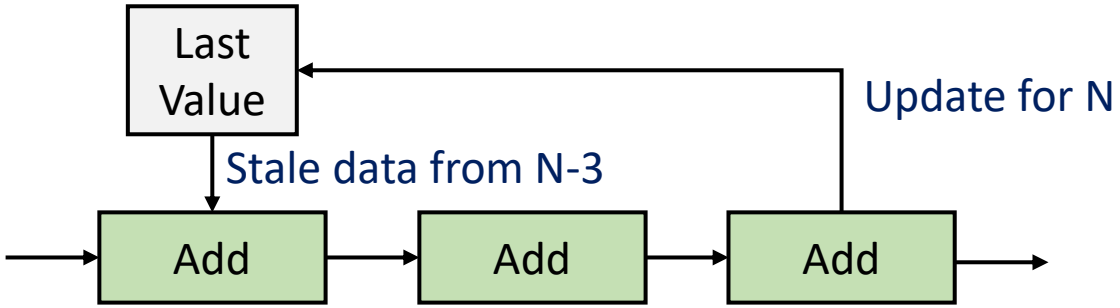
Datapath Width Bottleneck

- A single datapath with a small width can bottleneck the whole pipeline
  - Unless it is on a path taken rarely
  - In that case, other widths can be made a bit wider to statistically compensate



Latency And Duty Cycle

- We have a problem if 4 parallel additions have long critical path
  - Why? Perhaps routing, fan-out, etc, …
- "Last Value" needs to be read, and then updated in the same cycle
  - If we pipeline the adds, the next input tuple will work on stale data
- This issue happens a lot! Especially for computations requiring shifts…

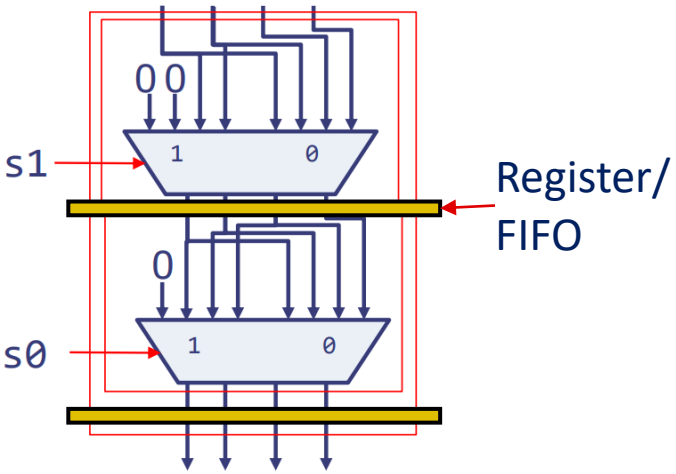


Latency And Duty Cycle

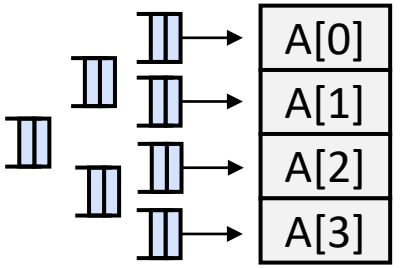
- For correct operation, pipeline must stall until dependency is resolved
  - In previous example, only one tuple processed per 3 cycles
  - Data throughput reduced to 1/3!
  - Since only one adder is used per cycle anyways, might as well only instantiate one adder and re-use that
- Duty cycle reduced to 1/3 …

Some Prominent Sources of Latency

- Remember: Variable-length shift has long critical path
- Solution: Pipelined shifts
  - Much smaller number of muxes (~1 typically) per cycle
  - But, now has high latency
  - Performance loss if dependency exists



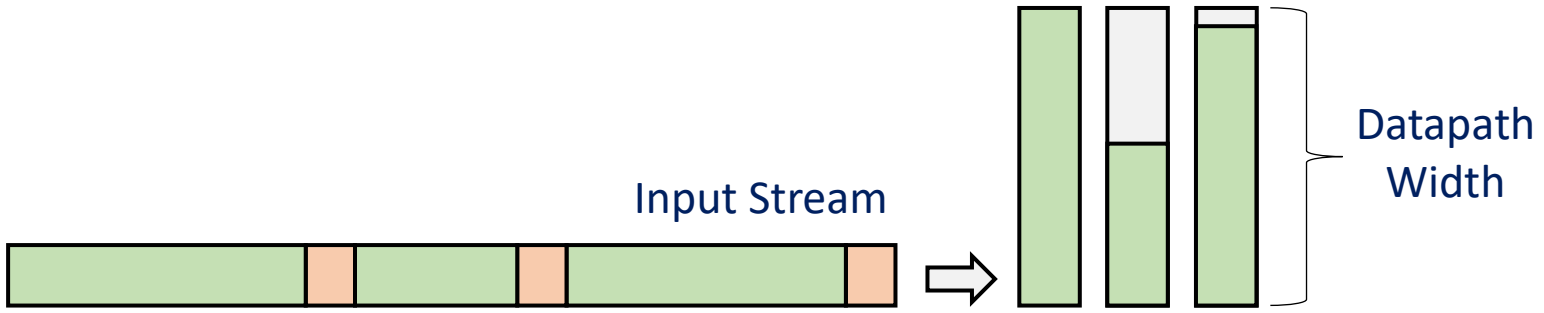
- Remember: Variable-index array access has long critical path
- Solution: Pipelined tree of scatter/gather
  - Much smaller number of muxes (~1 typically) per cycle
  - But, now has high latency
  - Performance loss if dependency exists

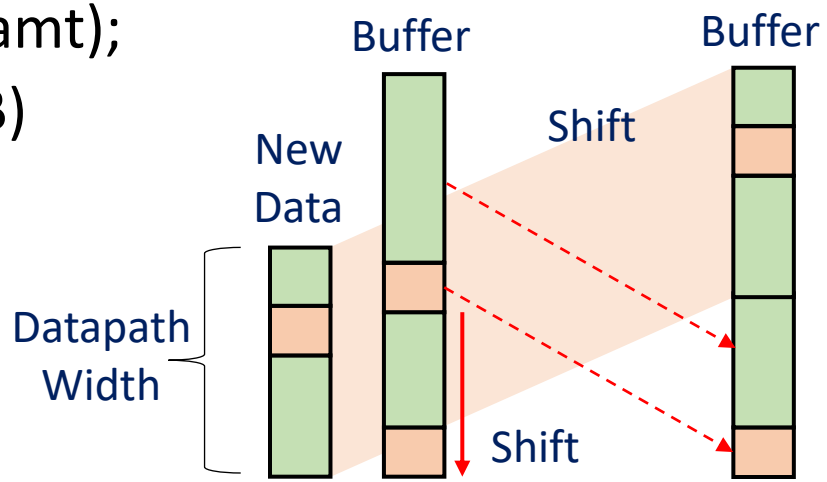


Like how we re-organized computation for wider datapaths, tricks can often be played to achieve both goals

Example: Variable Length Decoding

- Data packet consists of a header (length) and variable-length data
  - We want to process one data element per cycle
  - Assume maximum data length is smaller than the datapath
  - Common pattern in data compression/encoding/packetization
- Since we have a fixed data path width, simplest solution uses variable-length shifts "Buffer" creates a dependency, reducing performance
  - e.g., buffer <= (buffer>>shiftamt) | (newdata<<shiftamt);
  - Moves the next header to Lowest Significant Bit (LSB)



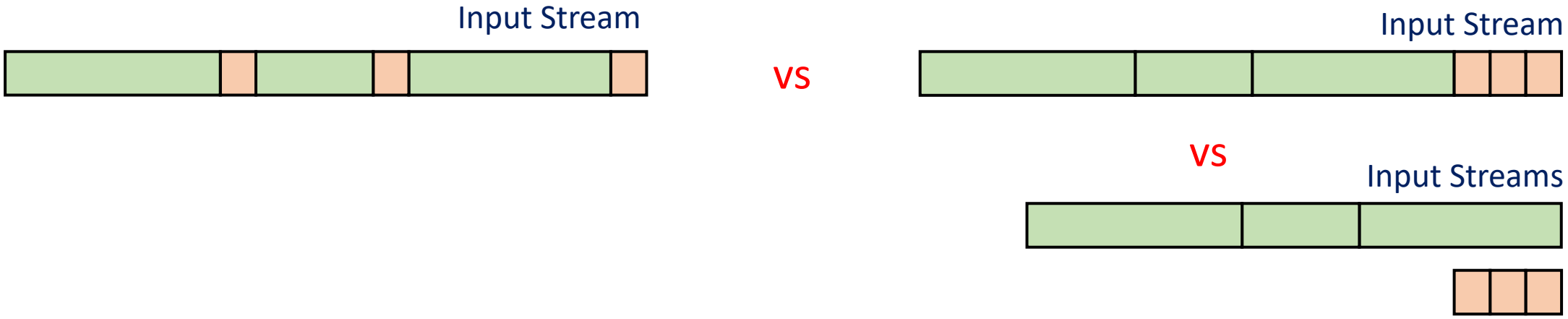


Example: Variable Length Decoding

- Two separate dependencies to handle
  - To do header decoding, header must be at LSB
  - Data must be “packed”, so that each data element is sent without internal gaps

Example: Variable Length Decoding

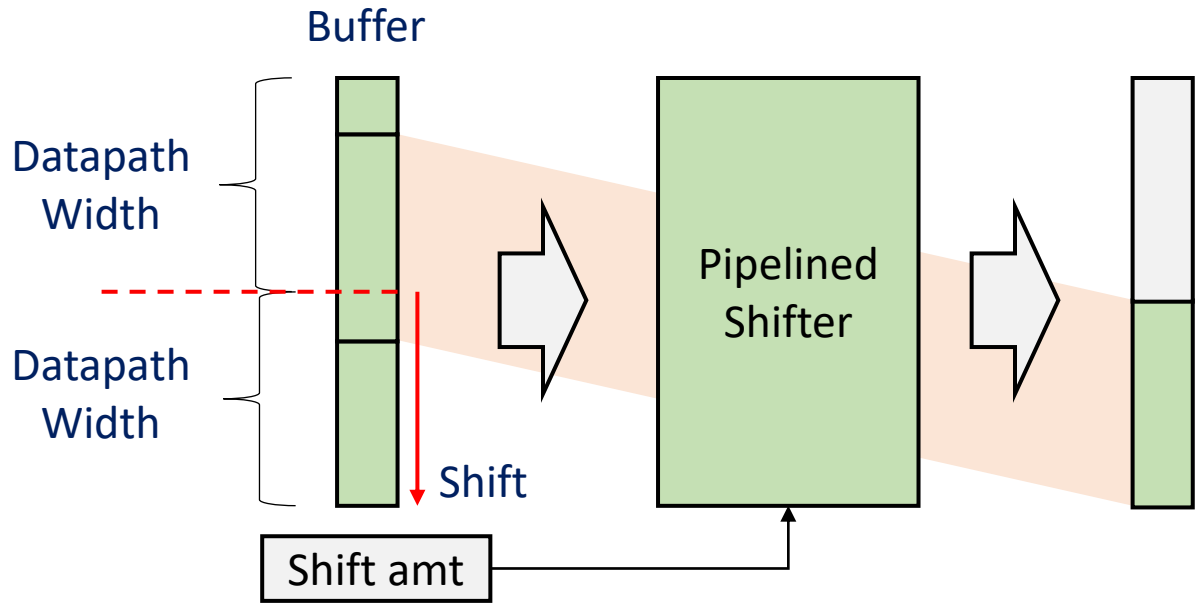
- Solution 1: Relax dependency for header decoding
  - May require data structure modification
  - e.g., group more headers together, or store headers separately



- Grouping headers can have more data in pipeline, but not always
  - Dependency between last element and first element in next tuple

Example: Variable Length Decoding

- Solution 2: Remove dependency on variable length shift
  - Instead of shifting right away, keep track of the accumulated shift amount
  - Shift amount given to pipelined shifter as parameter
  - If shift amount becomes larger than datapath width, subtract datapath width
    - And shift buffer down fixed amount, by datapath width



No more dependency on variable length shift

Performance of Accelerators

- Factors affecting accelerator performance
  - Throughput of the computation pipeline
    - How wide can we make the datapath?
    - Can it be replicated? (Amdahl's law? Chip resource limitations? Data dependency?)
  - Performance of the on-board memory
    - Is it sequential? Random? (Latency affects random access bandwidth for small memory systems with few DIMMs) – Effects may be order of magnitude
    - Latency bound?
  - Performance of host-side link (PCIe?)
    - Latency bound?
- Any factor may be the critical bottleneck

Aside: Data Rate And Deadlocks

- Remember: Two-phase memory access
  - On entity sends memory read requests, another receives data
- If multiple modules access memory, an arbiter is implemented
  - Tries to do fair scheduling between requesters
  - For efficiency, typically accepts burst requests
    - Burst size of 8+ KB makes best use of DRAM architecture (Row buffers)
- Multiple streams of data may be merged later
  - e.g., Three decompression cores working on separate columns, merged into rows
  - What happens when the compression rate / decompressed data rate is different?

Aside: Data Rate And Deadlocks

- Merging streams with different data rates may cause deadlocks
  - Data read for fast stream waiting in DRAM read queue
  - DRAM request queue is full because of the fast stream
  - Slow stream data not available for merging
- Solution: Flow control
  - Each fetcher needs to manage a buffer (large enough to hide memory latency)
  - Also keep track of how much data is in the buffer + how much request in flight
  - Send memory request if current buffer + in flight data is smaller than buffer size

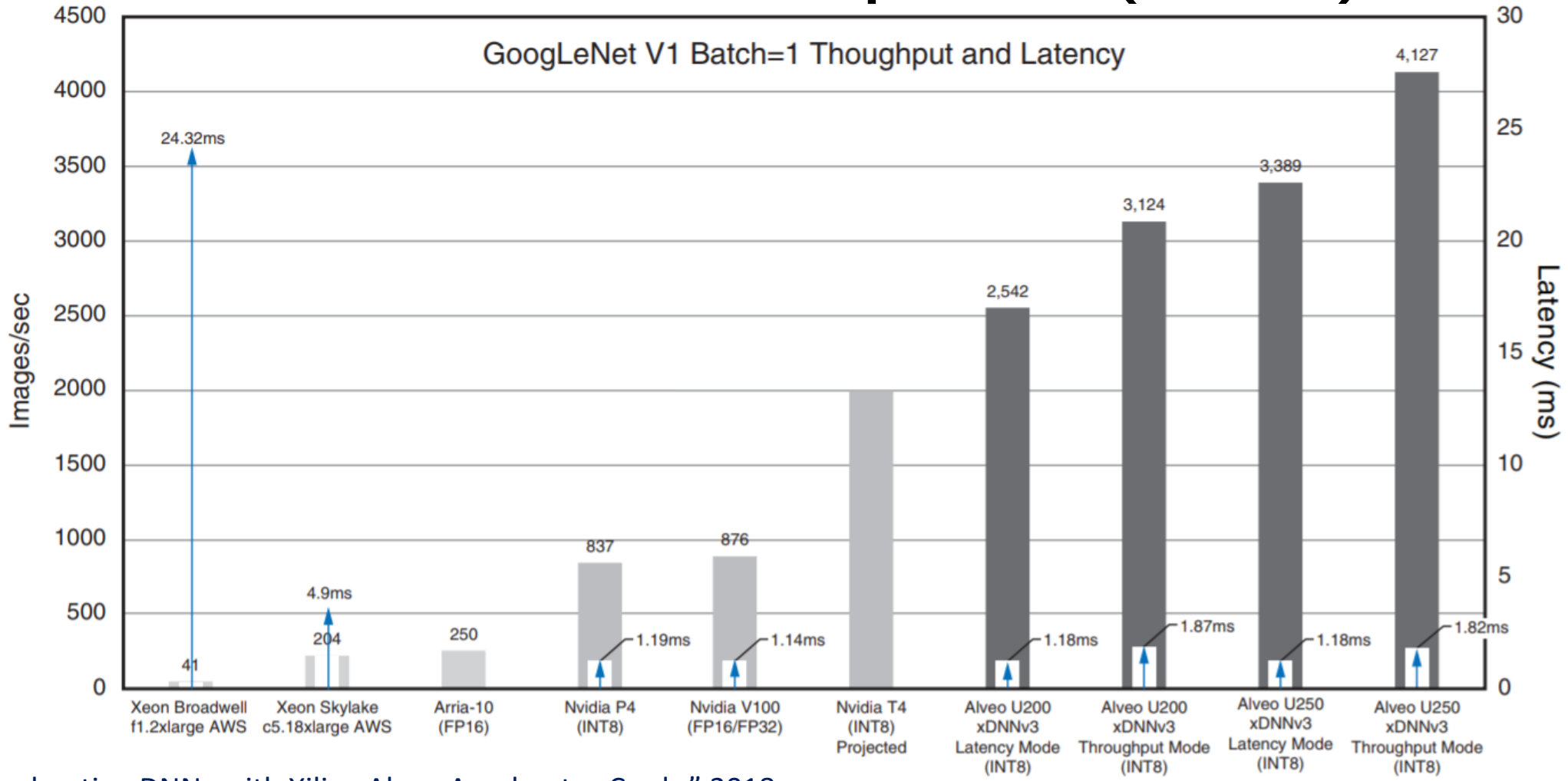
CS 250B: Modern Computer Systems

FPGA Accelerator Design Principles
Part 2: Timing Issues

Sang-Woo Jun

UCI

FPGA Performance Snapshot (2018)



Xilinx Inc., “Accelerating DNNs with Xilinx Alveo Accelerator Cards,” 2018

*Nvidia P4 and V100 numbers taken from Nvidia Technical Overview, "Deep Learning Platform, Giant Leaps in Performance and Efficiency for AI Services, from the Data Center to the Network's Edge

FPGA Performance and Clock Speed

- Typical FPGA clock speed is much slower than CPU/GPUs
  - Modern CPUs around 3 GHz
  - Modern GPUs around 1.5 GHz
  - FPGAs vary by design, typically up to 0.5 GHz
- Then how are FPGAs so fast?
  - Simple answer: Efficient fine-grained parallelism

- But before we jump into that topic… Some background!

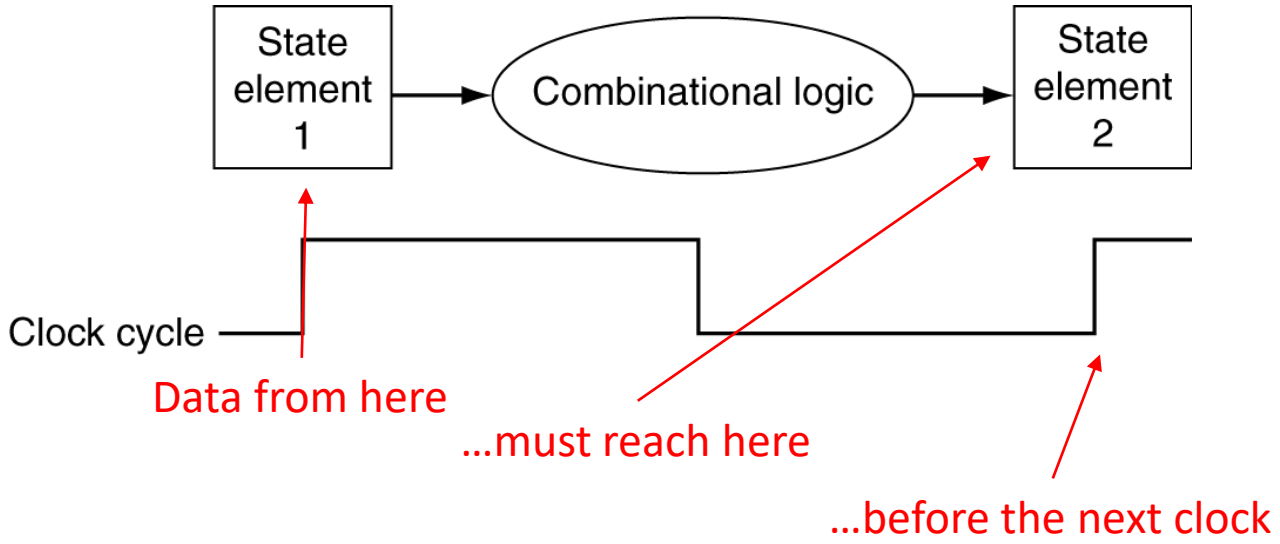
New Constraints in Hardware Design

- New from the perspective of software programmers

- Propagation delay and clock speed
  - What kind of logic has high or low propagation delay?
  - Logic depth
  - Placement/Routing
  - Fan-Out

Remember: Propagation Delay

- In a clock-synchronous sequential circuit, the “processing” (combinational logic) must fit in a clock cycle
  - Combinational logic has propagation delay, which must be less than the clock cycle
  - If not, clock speed must be lowered, losing performance!
- What kind of logic has high propagation delay?



Aside: Timing Violation Reports

- If there is a propagation delay that is too long, the synthesis tool will complain
  - Typically part of the synthesis report
  - e.g., “post_route_timing_summary.rpt” (Xilinx Vivado)

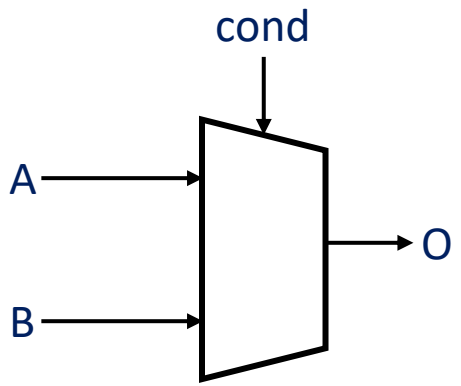
Aside: Timing Violation Reports

```
Max Delay Paths
--------------------------------------------------------------------------------
Slack (VIOLATED) :        -4.188ns  (required time - arrival time)
  Source:                 hwmain_[omitted]_pipe1_comp_d_cnt_reg[8]/C
                            (rising edge-triggered cell FDRE clocked by userclk2  {rise@0.000ns fall@2.000ns
 period=4.000ns})
  Destination:            hwmain_[omitted]_pipe1_dram_writer2_inq_memory/RAM_reg_12/DIPADIP[0]
                            (rising edge-triggered cell RAMB36E1 clocked by userclk2  {rise@0.000ns fall@2.0
00ns period=4.000ns})
  Path Group:             userclk2
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            4.000ns  (userclk2 rise@4.000ns - userclk2 rise@0.000ns)
  Data Path Delay:        7.314ns  (logic 2.756ns (37.682%)  route 4.558ns (62.318%))
  Logic Levels:           18  (CARRY4=10 LUT2=1 LUT3=3 LUT4=1 LUT6=1)
  Clock Path Skew:        -0.266ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    5.997ns = ( 9.997 - 4.000 )
    Source Clock Delay      (SCD):    6.727ns
    Clock Pessimism Removal (CPR):    0.464ns
  Clock Uncertainty:      0.065ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.108ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
```

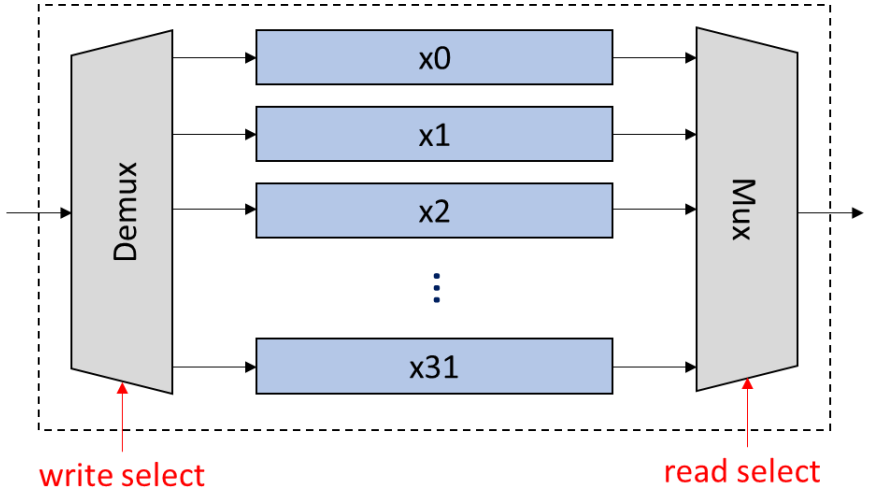
Factor #1: Logic Depth

- The most straightforward factor: Length of the critical path of the circuit
  - What kind of logic results in high propagation delay?
  - We need some insight to try to re-organize computation and remove timing violations
- One case: Simply, just too much logic in a single cycle/rule
  - e.g., 16-dimension Euclidean distance
    - Will 8 dimensions work?
  - Many nested if-else statements
    - Each conditional creates an expensive layer of multiplexers!
  - Multiplication is expensive, compared to add/sub



Anecdote: Hrishikesh et.al., "The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays." ISCA 2002

Factor #1: Logic Depth

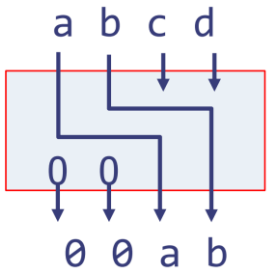


- Problem with very large FIFOs
  - e.g., mkSizedFIFO(64)
- FIFO semantics say something enqueued must be available next cycle
  - Combinational path from all elements to fifo.first
  - For many elements, many multiplexers!
- If latency is not a problem, consider chaining multiple FIFOs via rules
- Or, use embedded block RAM via mkSizedBRAMFIFO
  - BRAM has hardwired addressing logic in non-configurable silicon

Factor #1: Logic Depth

- Common pitfall: Variable-length shift
  - e.g., Given Bit#(32) b,s; computing b<<s is very expensive
- But, fixed-length shift is pretty much free

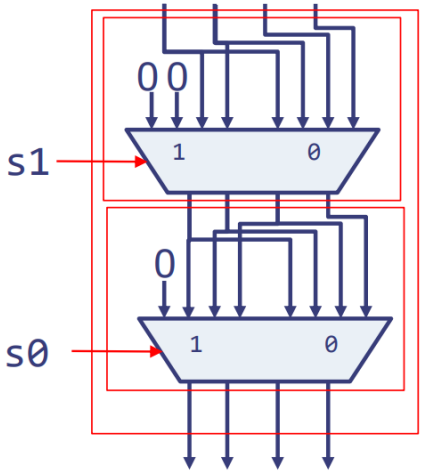
let a = b<<1;



Simple re-labeling of wires, no delay

but

let a = b<<s;

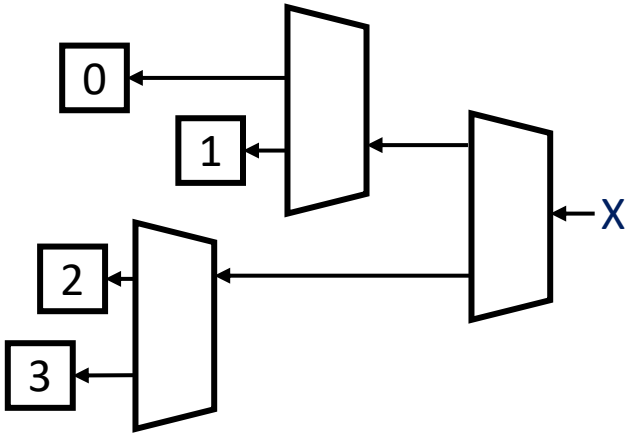


Instantiates barrel shifter!
Very large delay
(Especially if s has many bits)

Suddenly not meeting timing!

Solution introduced later!

Factor #1: Logic Depth



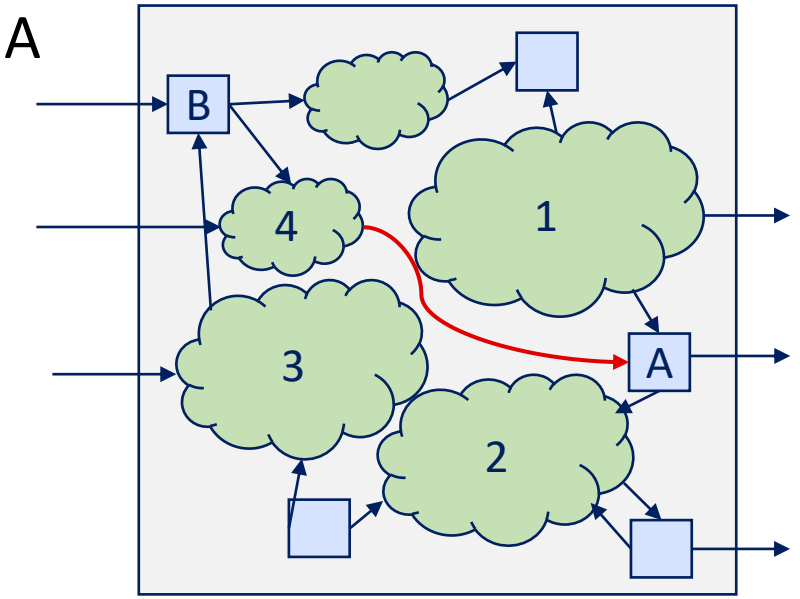
- Common pitfall: Variable index access
  - e.g., vector1[idx] <= x; //Where idx is a state variable (Reg#)
  - Creates a tree of multiplexers, very deep if vector/index is large

Solution introduced later!

- Common pitfall: Using modulo
  - e.g., if ((counter+1) % 32 == 0) begin …
  - Involves an expensive division operation
  - Replace with:
    if (counter + 1 == 32) begin
    counter <= 0; …
  - If power of two, use bit masks: if (counter & 32'b11111 == 32'b11111) begin …

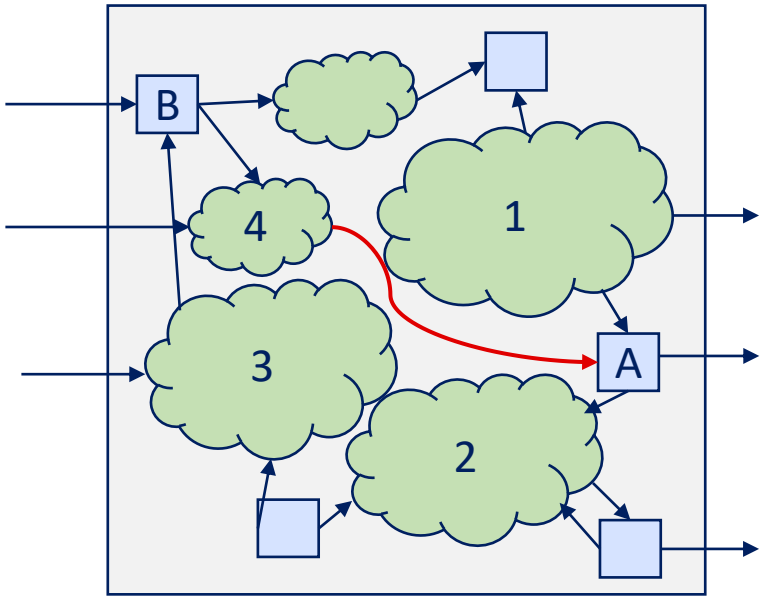
Factor #2: Placement And Routing

- Simplified introduction to placement/routing
  - Mapping state elements and combinational circuits to limited chip space
    - Done by the synthesis tool
    - May add significant propagation delay to combinational circuits
- Example:
  - Complex combinational circuits 1 and 2 accessing state A
  - Spatial constraints push combinational circuit 4 far from state A
  - Path from B to A via 4 is now very long!



Factor #2: Placement And Routing

- Rule of thumb:
  - One combinational circuit (rule) should not access too many states
  - One state should not be used by too many combinational circuits
- If state A can afford to be latency-tolerant, exploit it
  - If “4” only writes to A, and latency tolerant,
    Create a FIFO to act as intermediate step before writing
  - If distance is very long, many FIFOs chained via rules
    - If simple fifo.enq logic incurs a timing violation, this may be the case

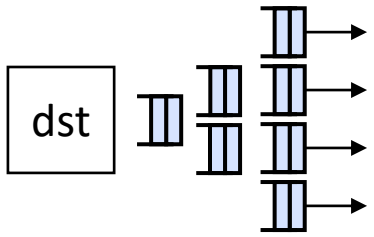


Factor #2: Placement And Routing

- A very wide bus may be difficult to route
  - All wires (for each bit) needs to start from the same source,
  - and arrive at the same destination within the same cycle
  - As chip becomes fuller, this may not always be possible
- Rule of thumb (anecdotally)
  - 256-bit busses at 250 MHz is around safe zone, for “normal”* computation

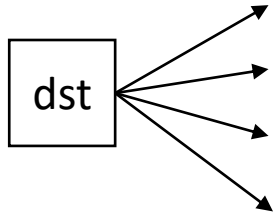
Factor #3: Fan-Out

- If a single register can be read by many circuits (High fan-out)
  - At circuit level, takes longer to generate enough charge to signal all destinations
- FPGAs handle high fan-out by the synthesis tool automatically generating buffers and replicated registers
  - However, still may add delay via the intermediate buffers
- Again, if register read/writes can be made latency-tolerant, do that!
  - e.g., Transfer reads to register via a tree of FIFOs



3 cycle latency, via 3 layers of rules

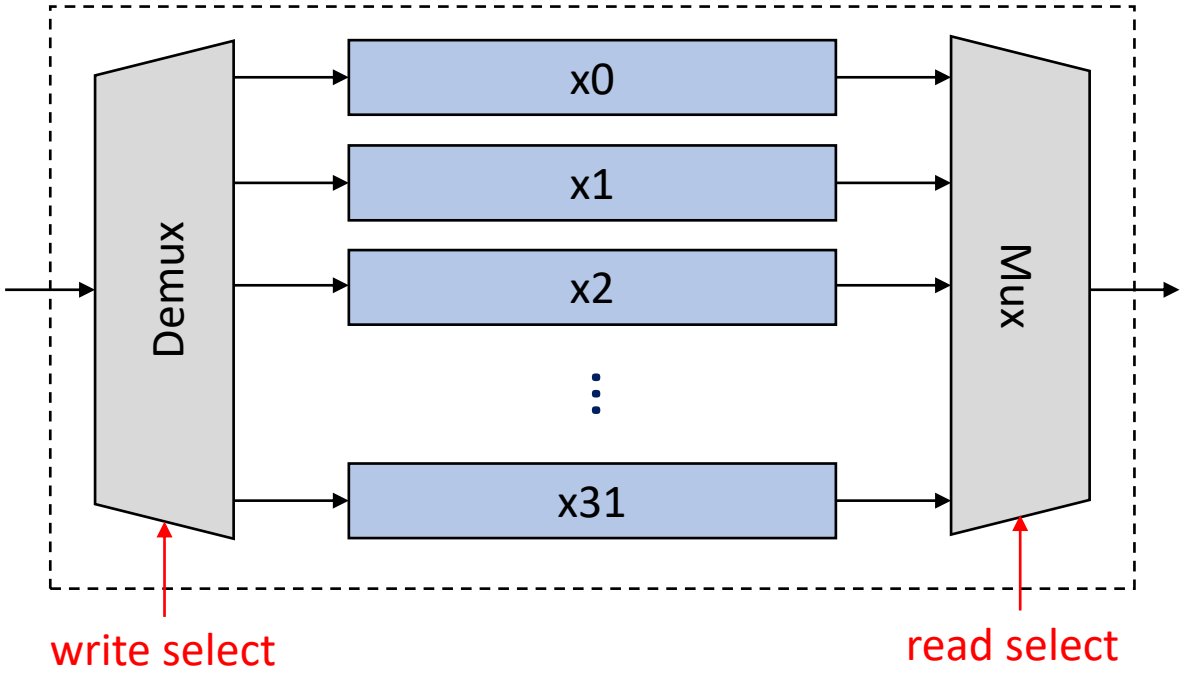
instead of

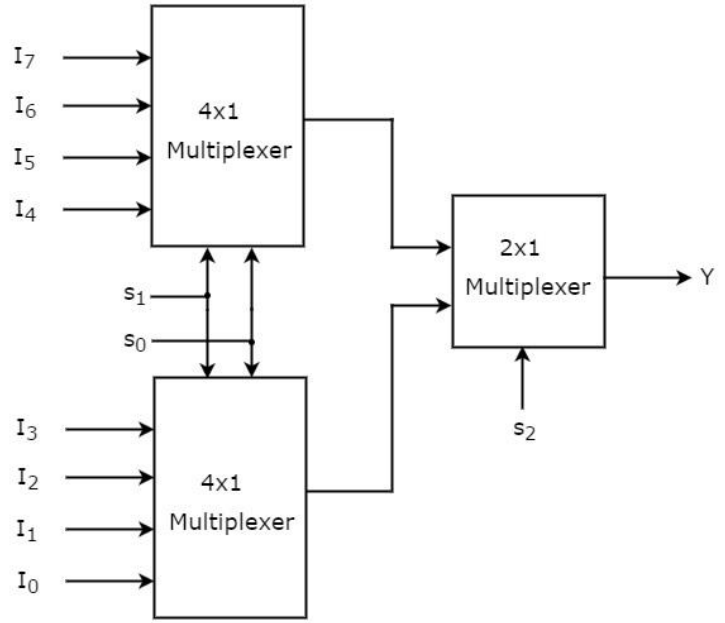


Fan-out of 4

Looking Back: Why Are Processor Register Files Small?

- Why are register files 32-element? Why not 1024 or more?





Hierarchical design of a 8x1 multiplexer

Propagation delay increases with more registers!