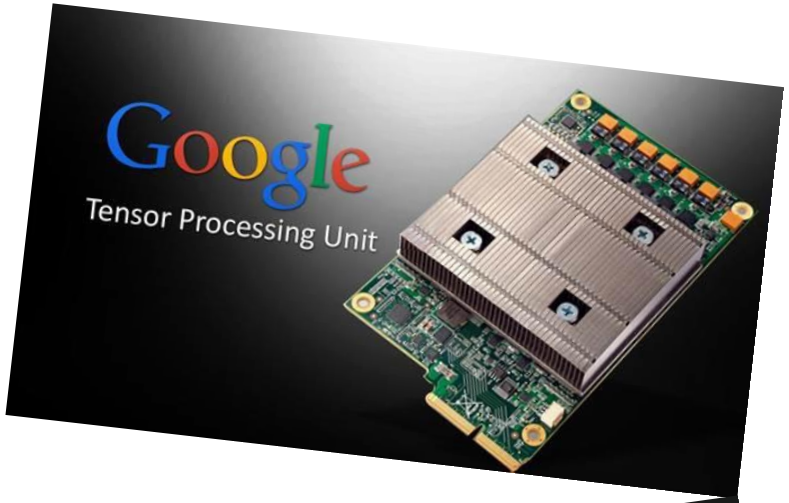
From Software Programs to Digital Circuits

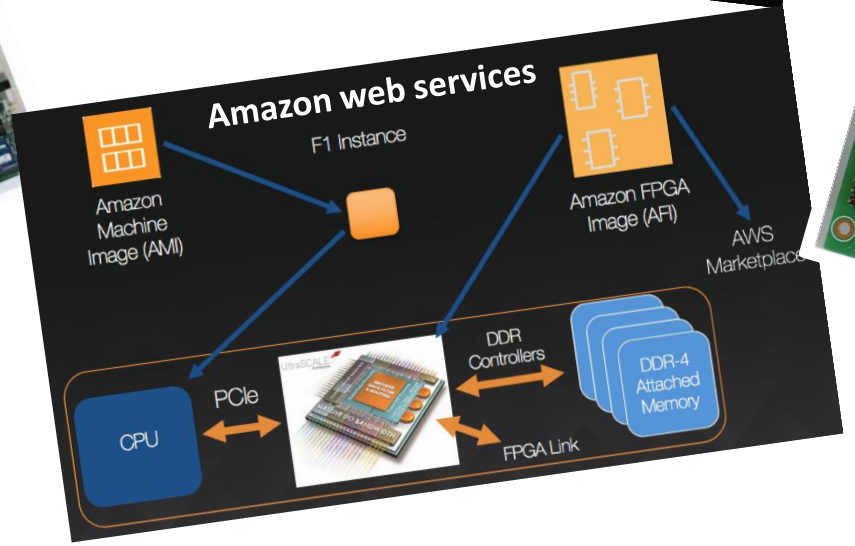
Lana Josipović

September 2023

TECHNOLOGY, MEDIA & TELECOM - INNOVATION JUNE 1, 2015 / 2:38 PM / UPDATED 7 YEARS AGO

Intel to buy Altera for $16.7 billion in its biggest deal ever





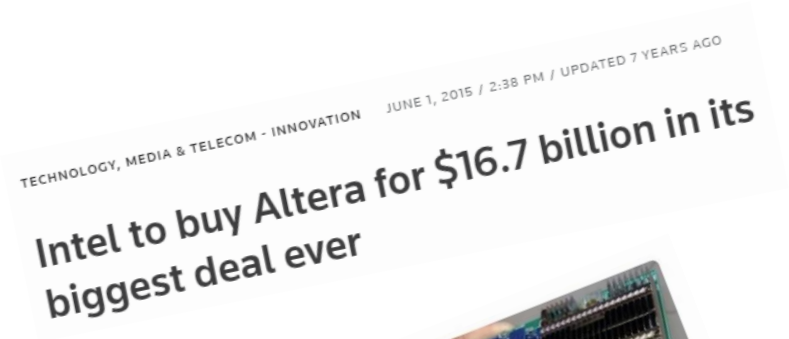
BUSINESS

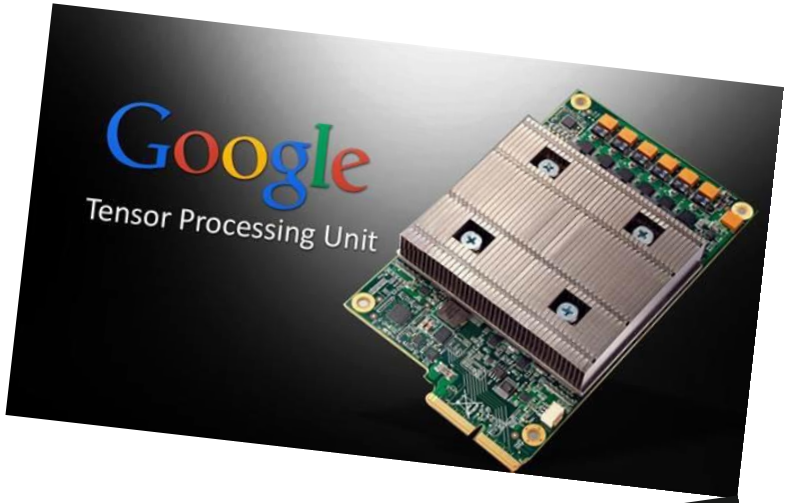
AMD Completes Record ~$50 Billion Acquisition of Xilinx

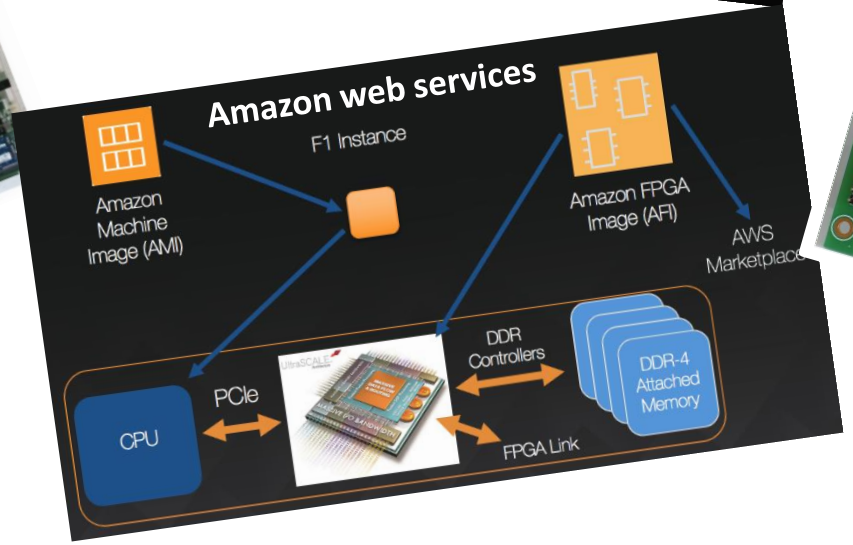
The largest deal in semiconductor history promises exciting times for FPGA technology.

Max Smolaks | Feb 14, 2022

Hardware acceleration for high parallelism and energy efficiency







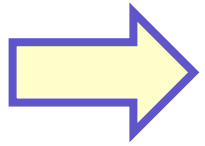
How to perform hardware design?

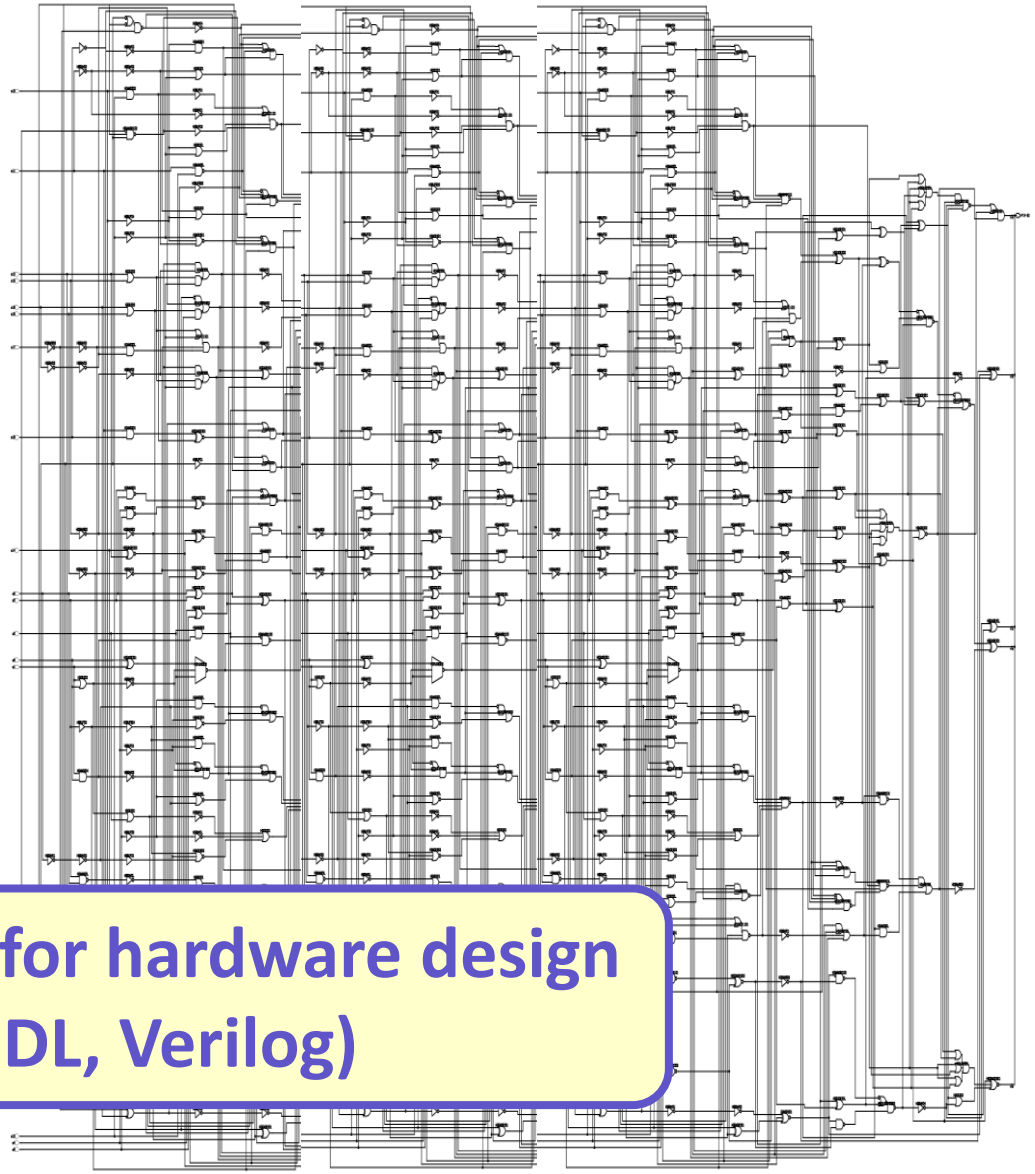
*… circuit design is often considered a **"black art",** restricted to only those with years of training in electrical engineering…*
[cacm.acm.org/magazines/2023/1/]

*… chips take **years to design**, resulting in the need to **speculate about how to optimize** the next generation of chips…*
[ai.googleblog.com/2020/04]

High-Level Synthesis: From Programs to Circuits

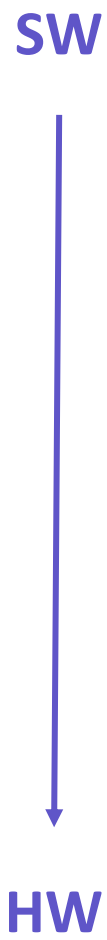
```
#define PI 3.14159265358979323846264338

complex* DFT_naive(complex* x, int N) {
  complex* X = (complex*) malloc(sizeof(struct complex_t) * N);
  int k, n;
  for(k = 0; k < N; k++) {
    X[k].re = 0.0;
    X[k].im = 0.0;
    for(n = 0; n < N; n++) {
      X[k] = add(X[k], multiply(x[n],
                                conv_from_polar(1,
                                                -2*PI*n*k/N)));
    }
  }

  return X;
}
```





Raise the level of abstraction for hardware design beyond RTL level (VHDL, Verilog)

Bridging the Gap Between Software and Hardware



Bridging the Gap Between Software and Hardware

SW

HW

HLS is still not meant
for software programmers

Bridging the Gap Between Software and Hardware

SW

HW

HLS is still not meant for software programmers

```
1  void(int* mem){
2    mem[512] = 0;
3    for(int i=0; i<512; i++)
4       mem[512] += mem[i];
5  }
```

(a) Unoptimized HLS Program; Execution Time = 27,236 clock cycles

Bridging the Gap Between Software and Hardware

SW

HW

HLS is still not meant for software programmers

```
1  void(int* mem){
2    mem[512] = 0;
3    for(int i=0; i<512; i++)
4       mem[512] += mem[i];
5  }
```

(a) Unoptimized HLS Program; Execution Time = 27,236 clock cycles

```
1  // Width of MPort = 16 * sizeof(int)
2  #define ChunkSize (sizeof(MPort)/sizeof(int))
3  #define LoopCount (512/ChunkSize)
4  // Maximize data width from memory
5  void(MPort* mem){
6     // Use a local buffer and burst access
7     MPort buff[LoopCount];
8     memcpy(buff, mem, LoopCount);
9     // Use a local variable for accumulation
10    int sum=0;
11    for(int i=1; i<LoopCount; i++){
12    // Use additional directives where useful
13    // e.g. pipeline and unroll for parallel exec.
14    #pragma PIPELINE
15       for(int j=0; j<ChunkSize; j++){
16       #pragma UNROLL
17          sum+=(int)(buff[i]>>j*sizeof(int)*8);}}
18     mem[512]=sum;
19 }
```

(b) Optimized HLS Program; Execution Time = 302 clock cycles

George et al. FPL 2014.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant
for software programmers

HLS often fails in extracting
parallelism from software code

HW

Bridging the Gap Between Software and Hardware

SW

HW

HLS is still not meant
for software programmers

HLS often fails in extracting parallelism from software code

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Sparse-matrix dense-vector multiplication (SpMV)

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

HW

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];
  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable memory latency

Variable loop bounds

Irregular memory access patterns

Sparse-matrix dense-vector multiplication (SpMV)

Bridging the Gap Between Software and Hardware

SW

HW

HLS is still not meant
for software programmers

HLS often fails in extracting
parallelism from software code

**HLS circuits need hardware-level
functional verification**

Bridging the Gap Between Software and Hardware

SW

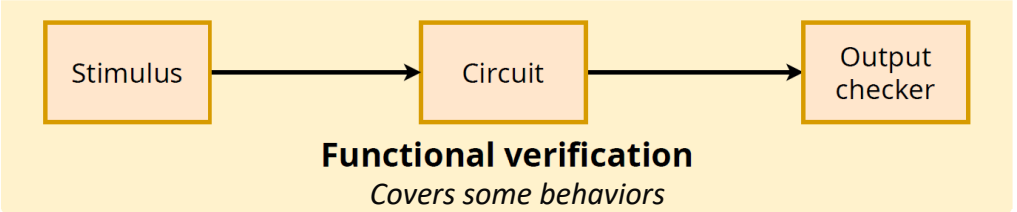
HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

HW

Functional verification of circuits using hardware simulation
→ inefficient, limited, non-exhaustive



Bridging the Gap Between Software and Hardware

SW

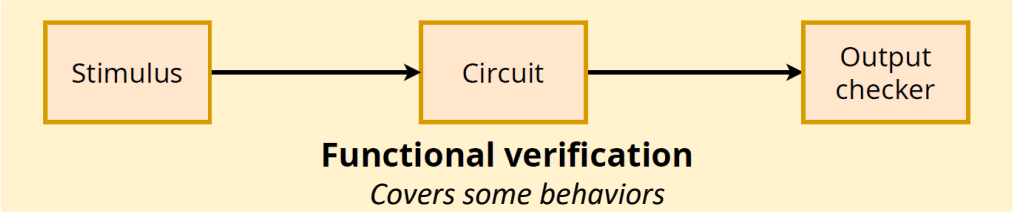
HLS is still not meant
for software programmers

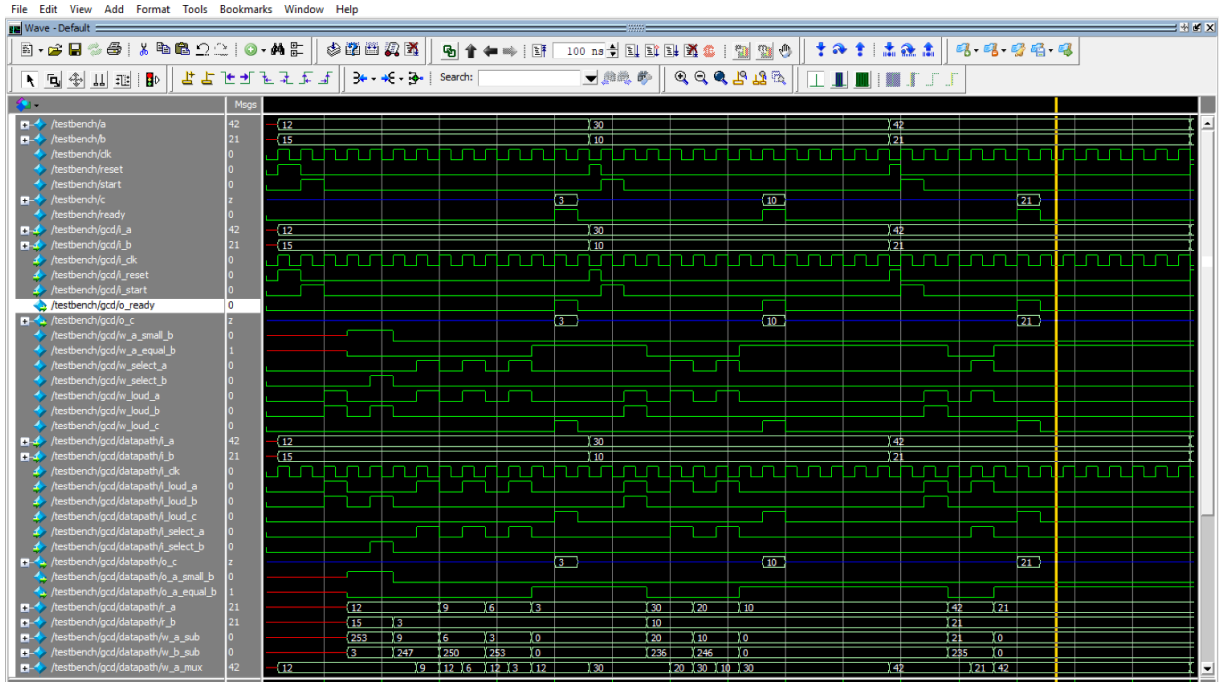
HLS often fails in extracting
parallelism from software code

**HLS circuits need hardware-level
functional verification**

HW

Functional verification of circuits using hardware simulation
→ inefficient, limited, non-exhaustive





Bridging the Gap Between Software and Hardware

SW

HLS is still not meant
for software programmers

HLS often fails in extracting
parallelism from software code

HLS circuits need hardware-level
functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

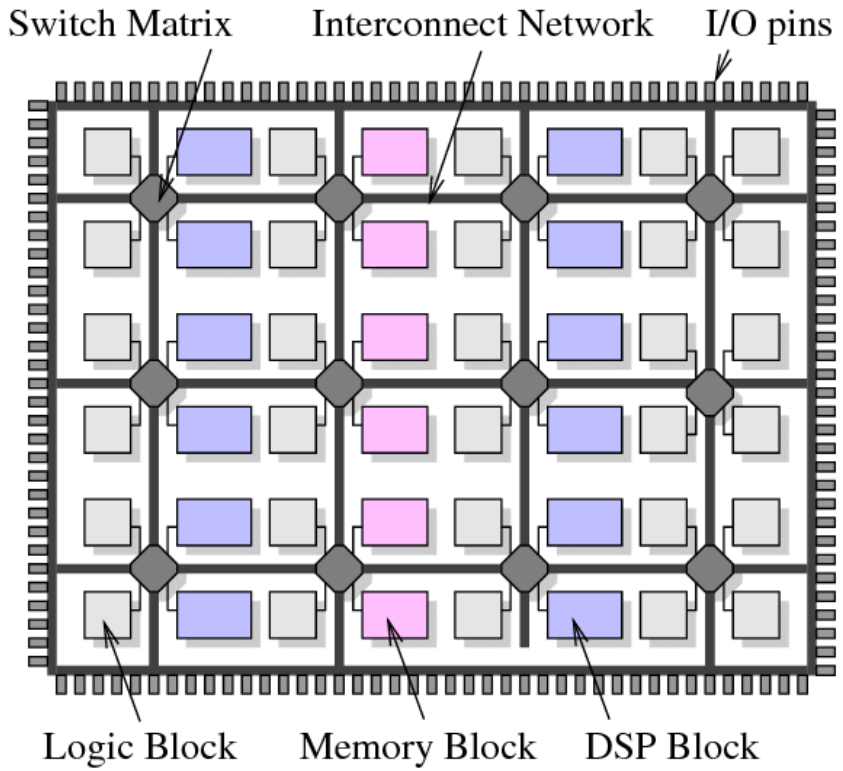
HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

FPGA technology mapping, placement, and routing
→ impact on circuit **performance and power**



Langhammer et al. ARITH 2015.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

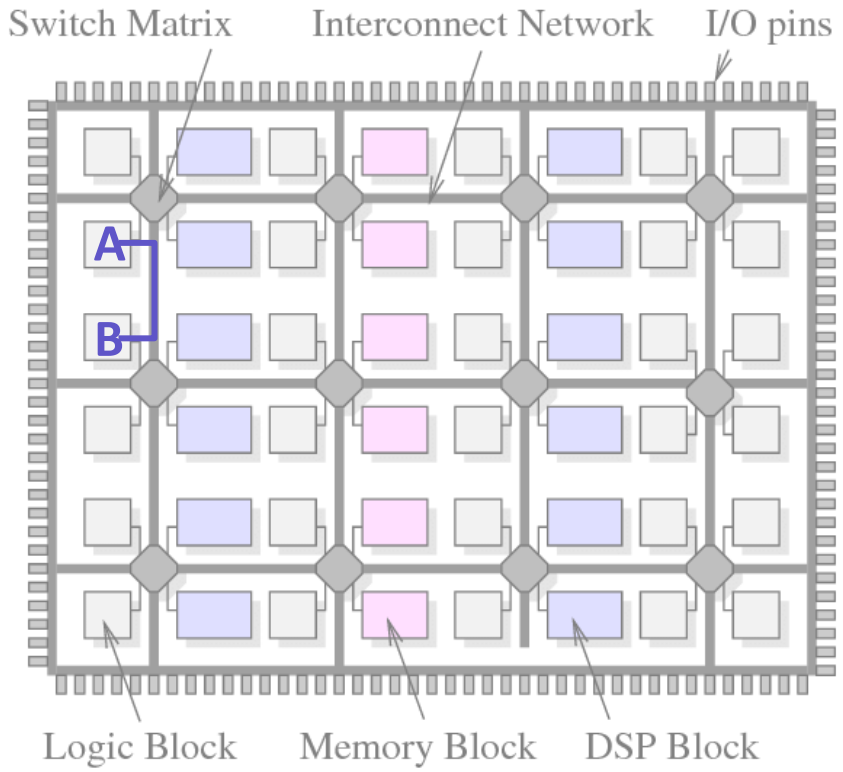
HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

FPGA technology mapping, placement, and routing
→ impact on circuit **performance and power**



A → B

Langhammer et al. ARITH 2015.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

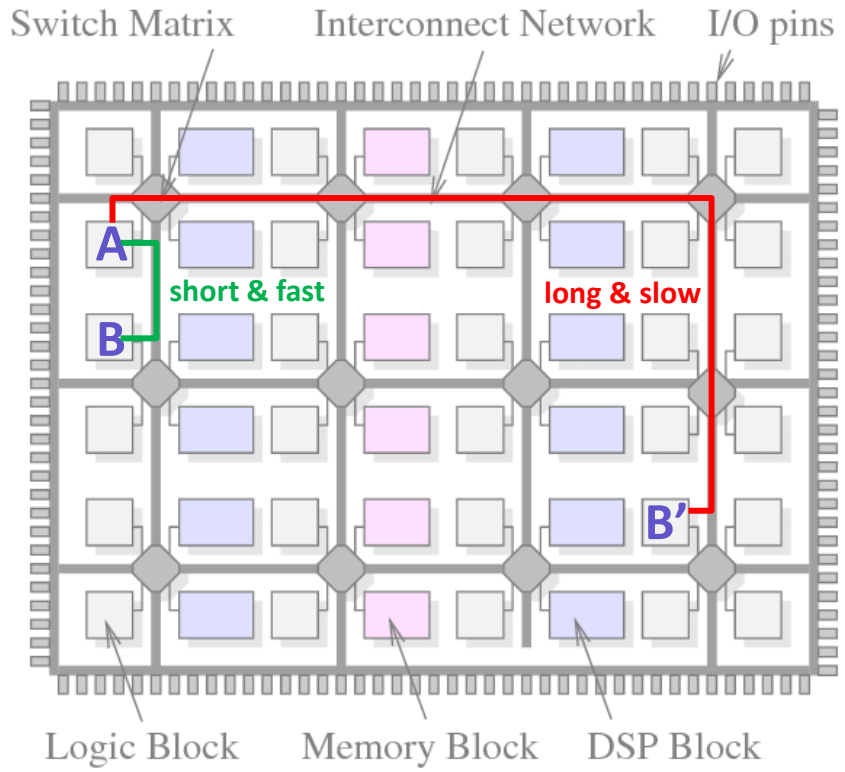
HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

FPGA technology mapping, placement, and routing
→ impact on circuit **performance and power**



A → B

Langhammer et al. ARITH 2015.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant
for software programmers

HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level
functional verification

It is difficult for HLS to account for
reconfigurable platform details

HW

How to generate high-performance circuits from general-purpose software code?

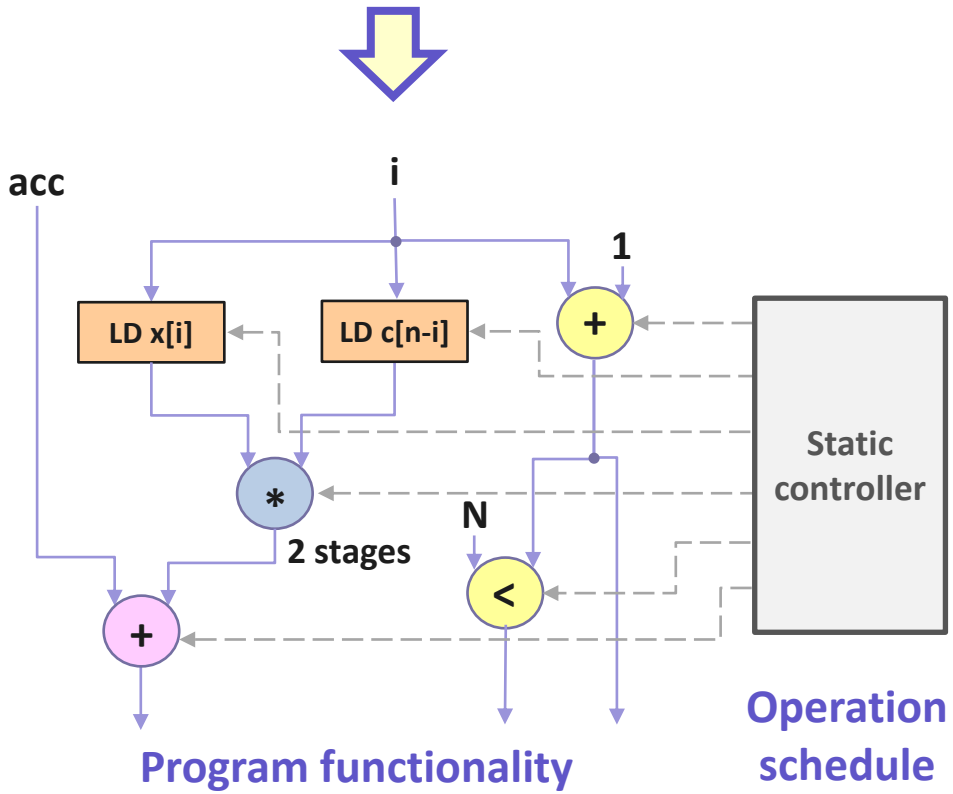
Standard HLS

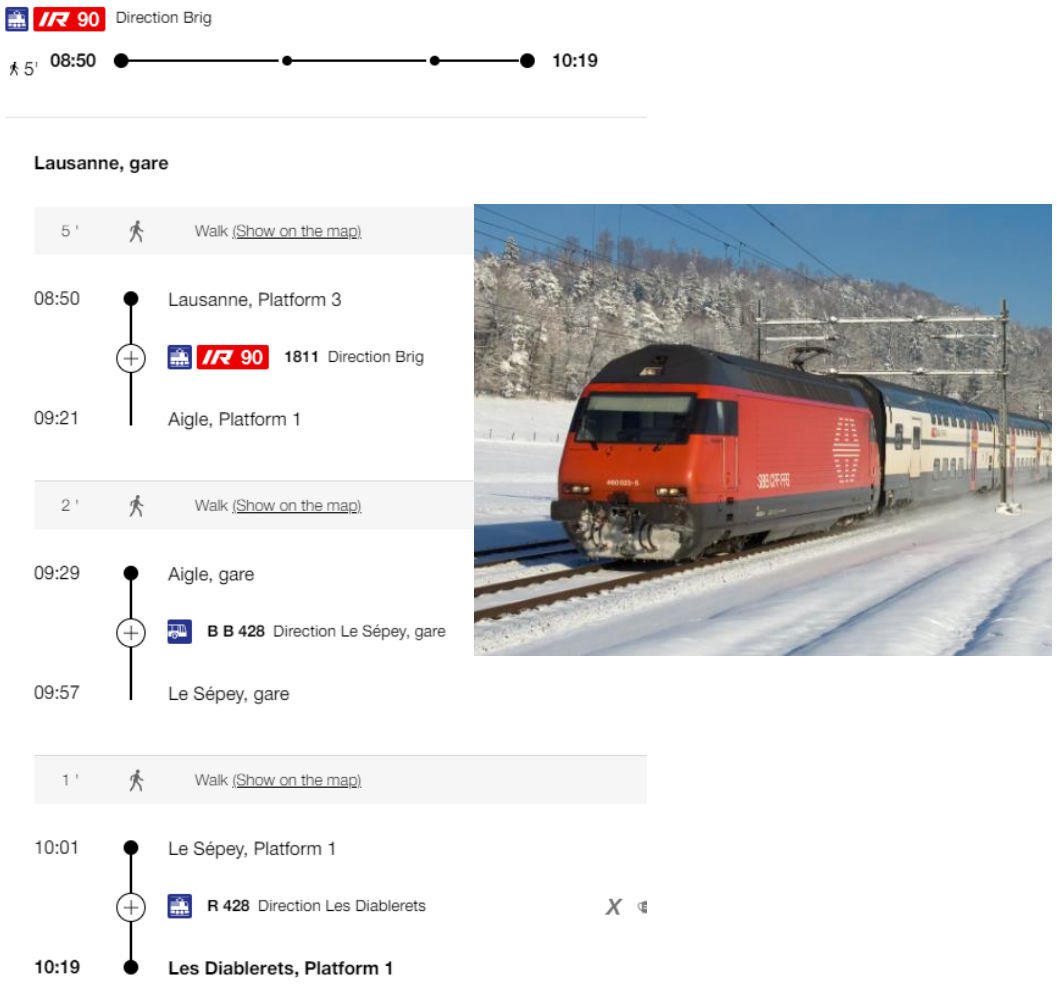
- **Create a datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

Standard HLS

- **Create a datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

```
for (i=0; i<n; i++) {
        acc += x[i] * c[n-i];
}
```

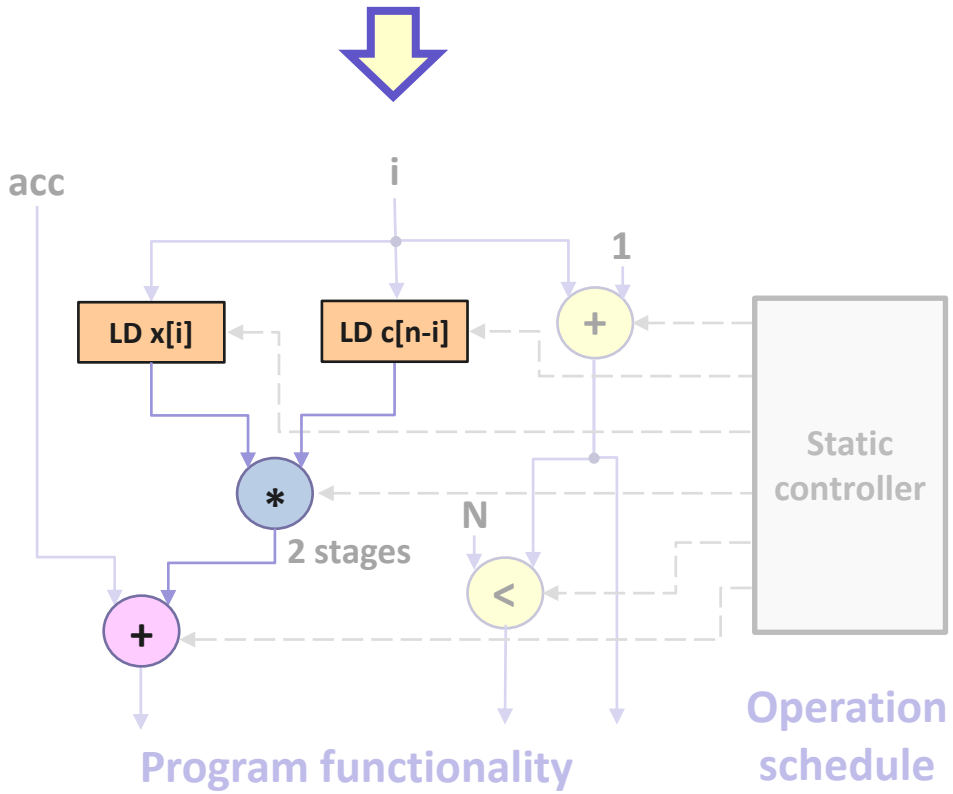


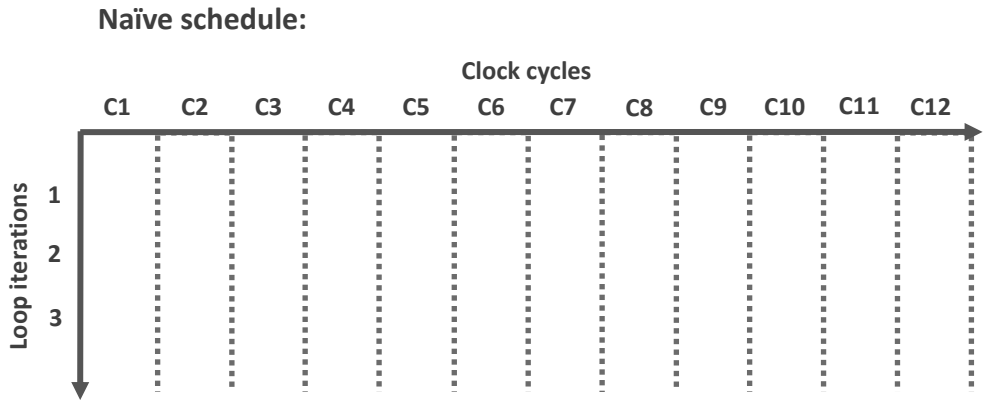


Standard HLS

- **Create a datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

```
for (i=0; i<n; i++) {
        acc += x[i] * c[n-i];
}
```

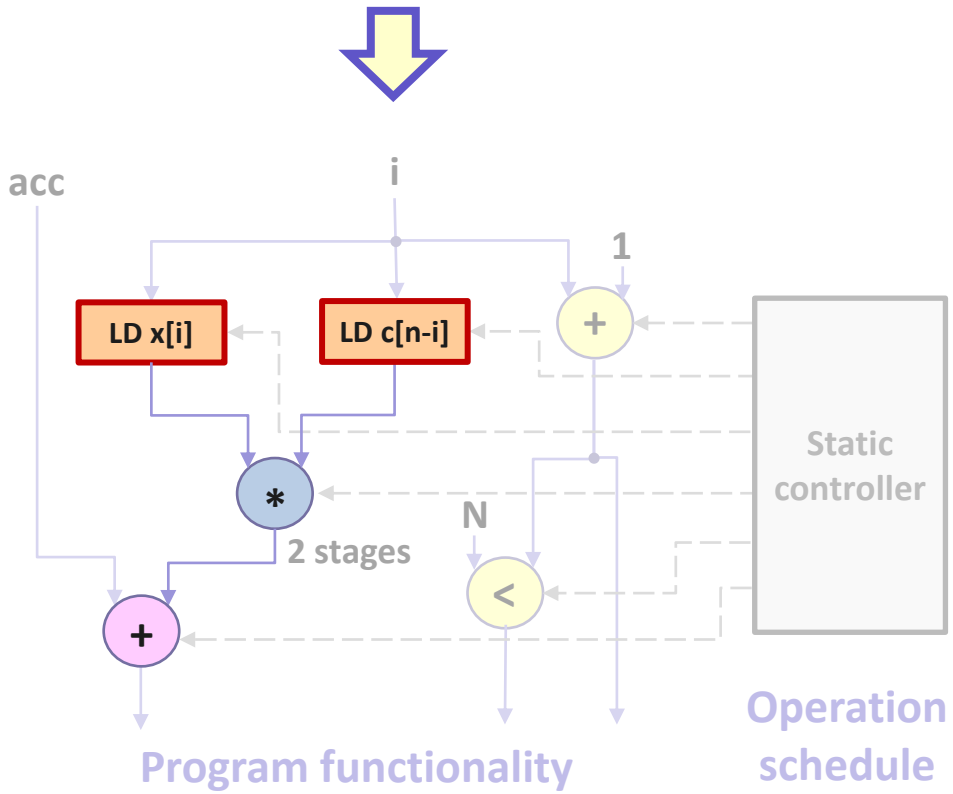


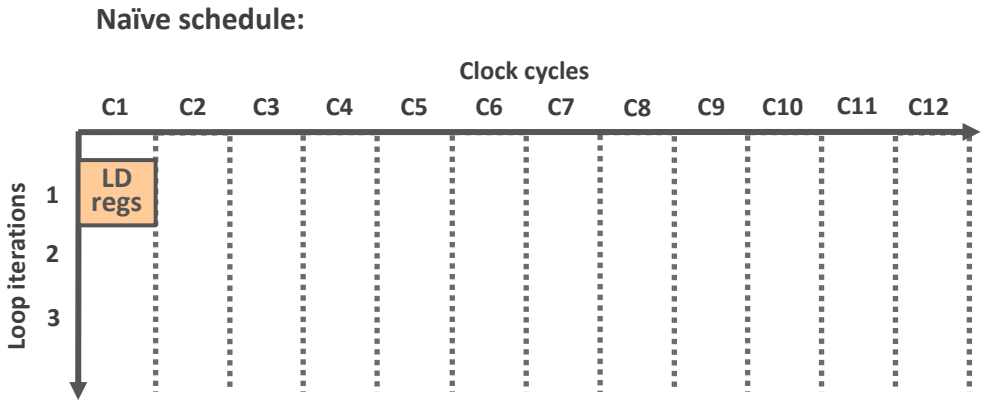


Standard HLS

- **Create a datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

```
for (i=0; i<n; i++) {
        acc += x[i] * c[n-i];
}
```

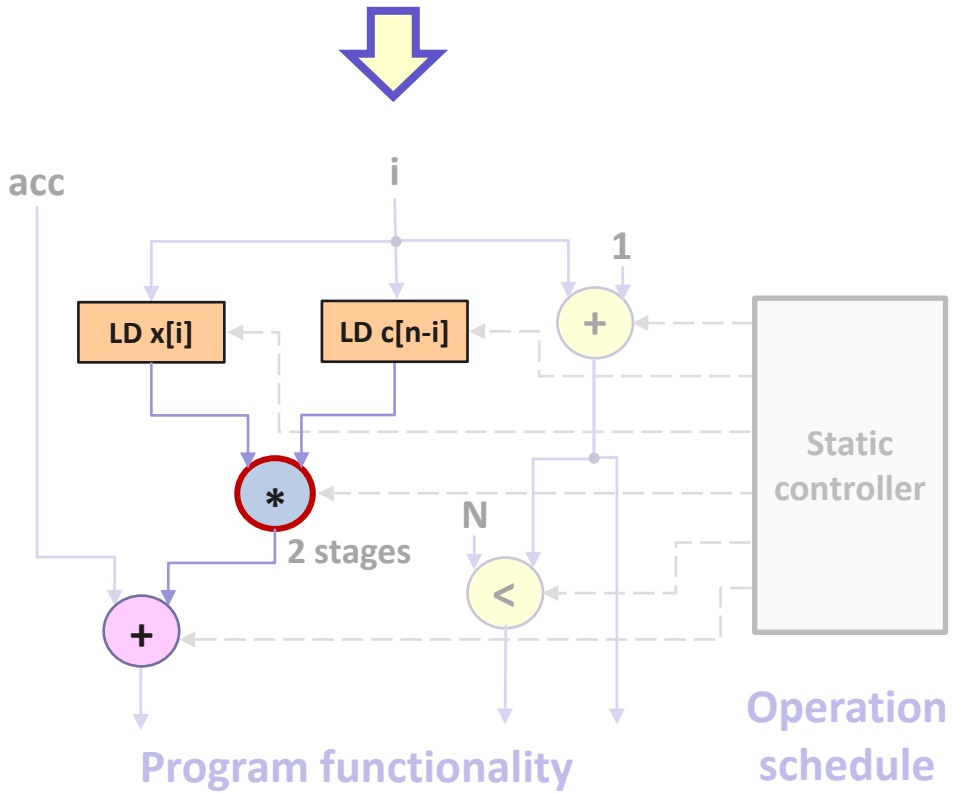


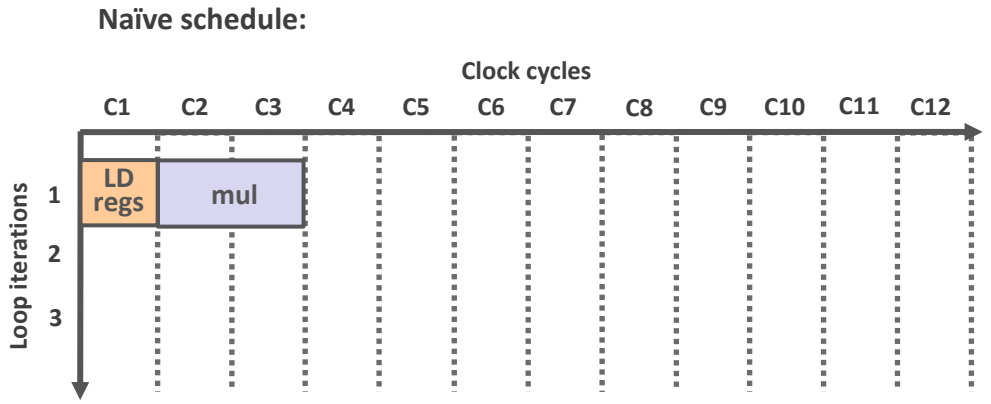


Standard HLS

- **Create a datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

```
for (i=0; i<n; i++) {
        acc += x[i] * c[n-i];
}
```

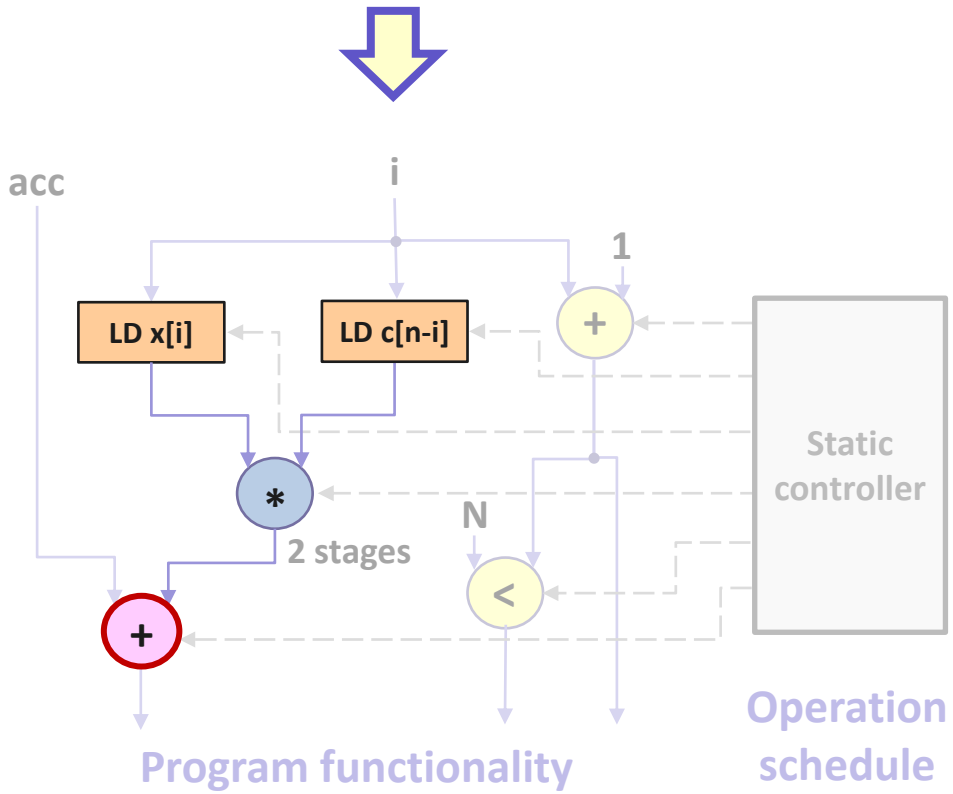


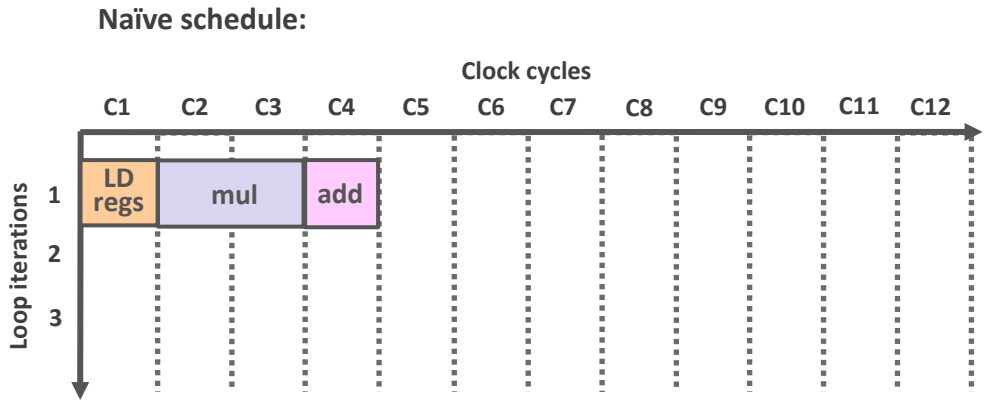


Standard HLS

- **Create a datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

```
for (i=0; i<n; i++) {
        acc += x[i] * c[n-i];
}
```

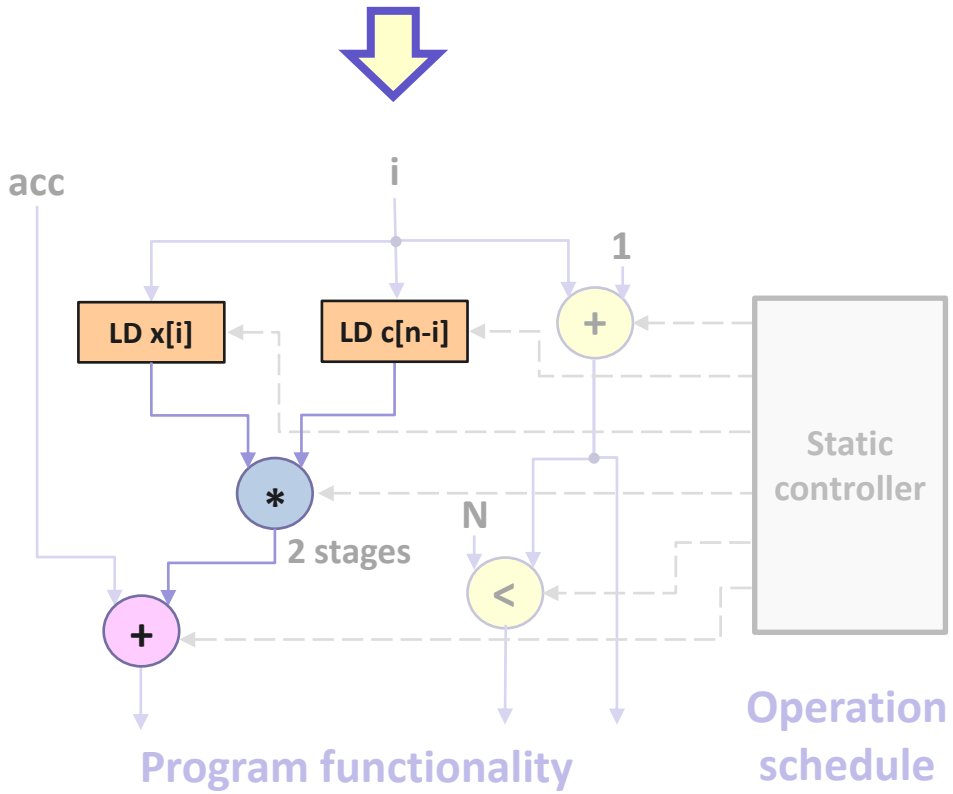




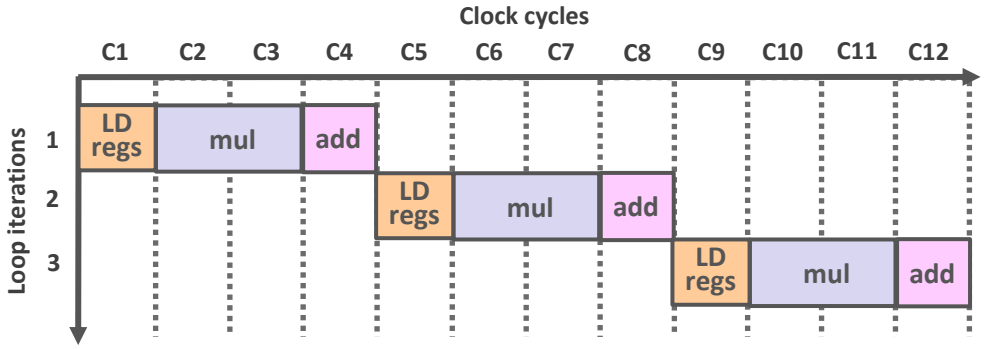
Standard HLS

- **Create a datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

```
for (i=0; i<n; i++) {
        acc += x[i] * c[n-i];
}
```



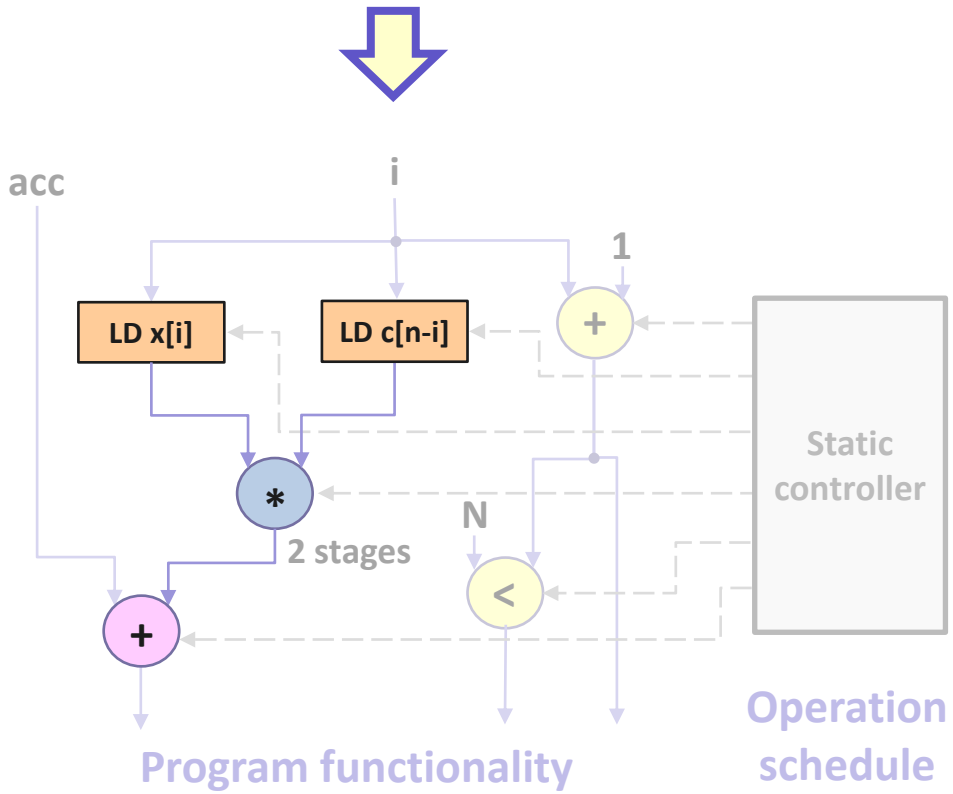
Naïve schedule:

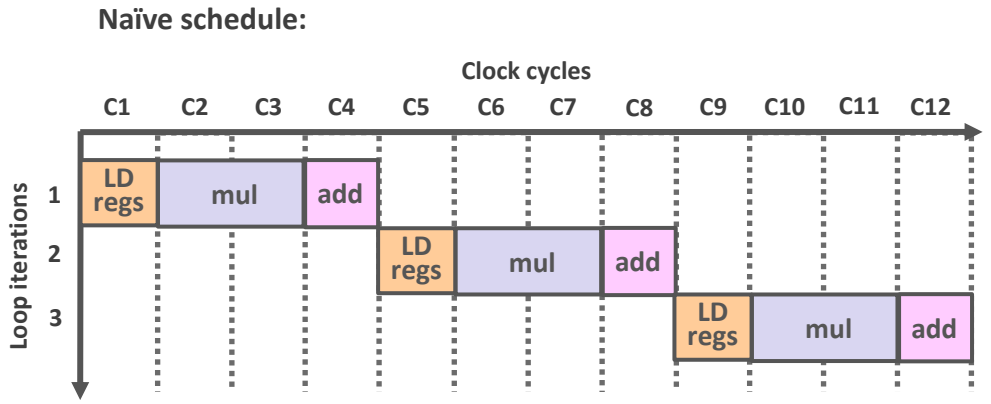


Standard HLS

- **Create a datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

```
for (i=0; i<n; i++) {
        acc += x[i] * c[n-i];
}
```



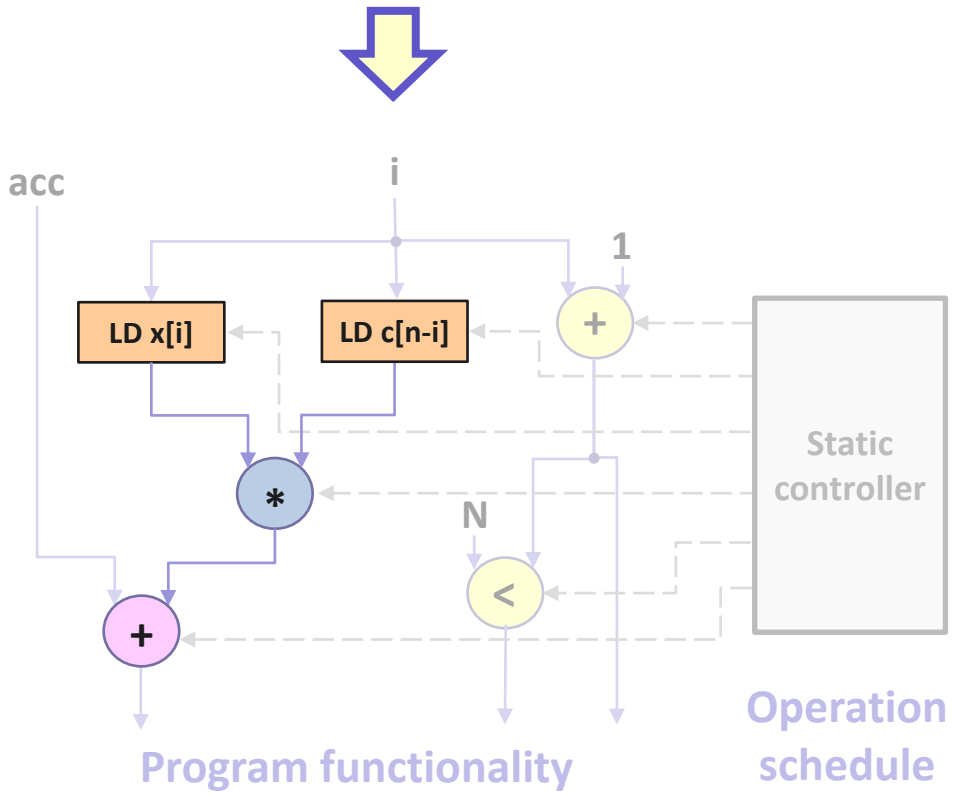


Low throughput: slow execution

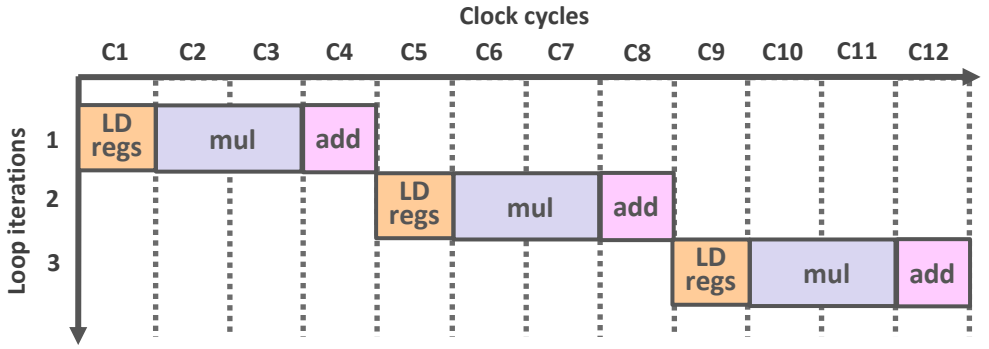
Standard HLS

- **Create a datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

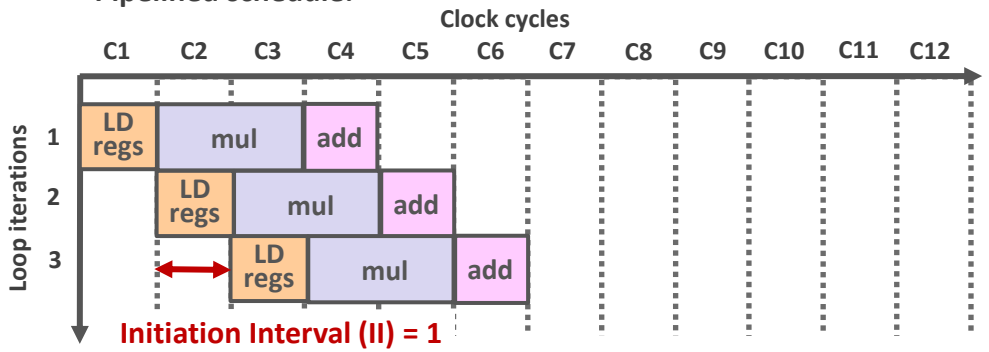
```
for (i=0; i<n; i++) {
        acc += x[i] * c[n-i];
}
```



Naïve schedule:



Pipelined schedule:



High throughput: fast execution

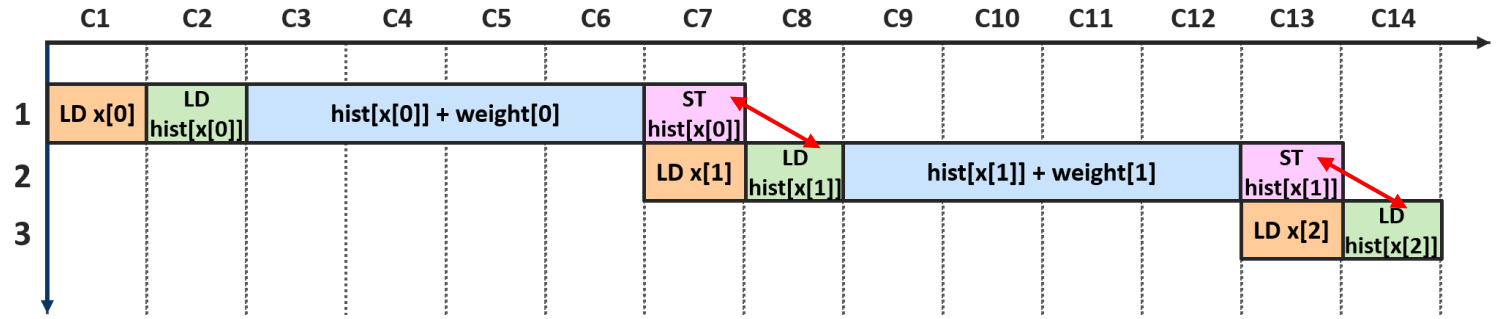
The Limitations of Static Scheduling

```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

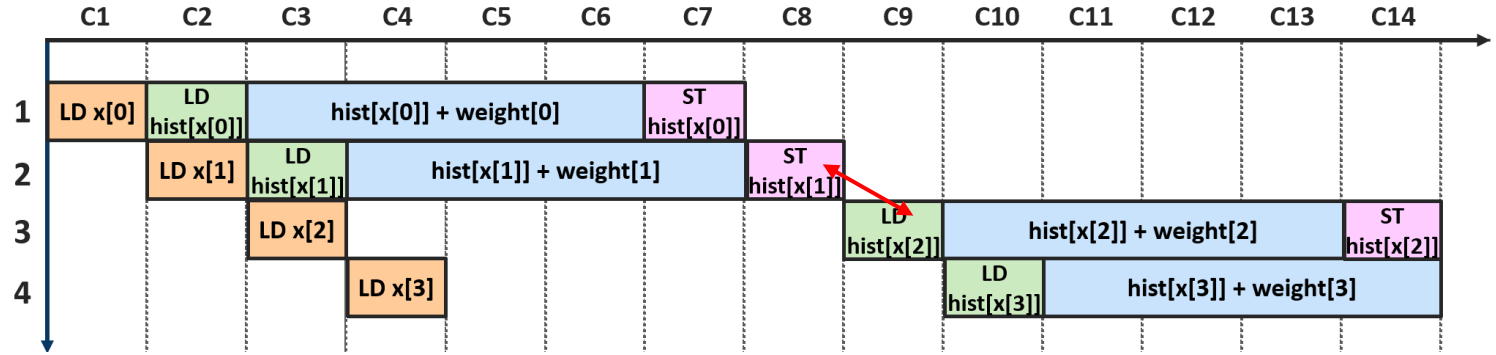
```
1: x[0]=5 → ld hist[5]; st hist[5];
2: x[1]=4 → ld hist[4]; st hist[4];
3: x[2]=4 → ld hist[4]; st hist[4];
```

RAW dependency

- Static scheduling (standard HLS tool)
  - Inferior when memory accesses cannot be disambiguated at compile time

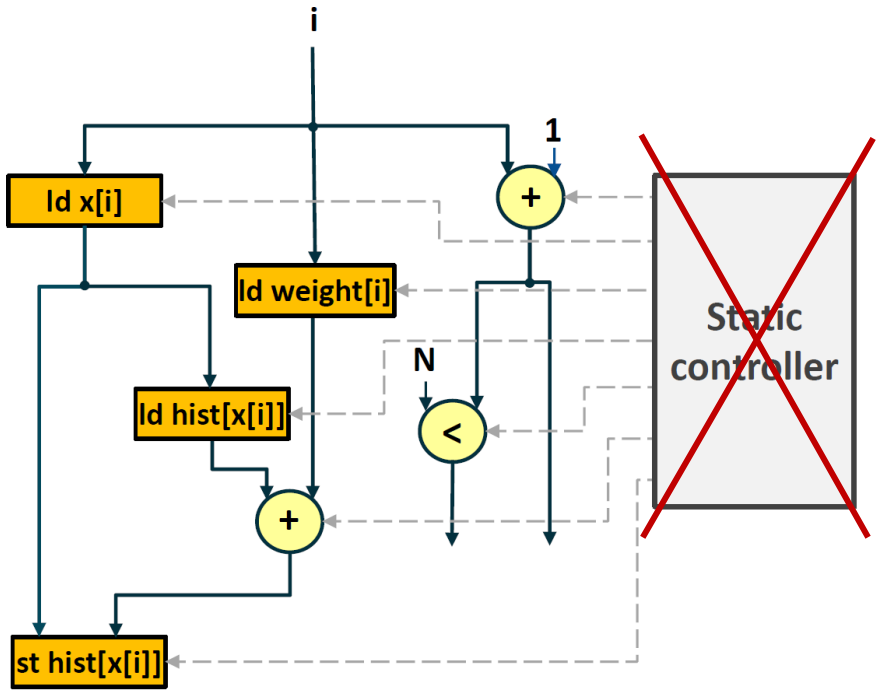


- Dynamic scheduling
  - Maximum parallelism: Only serialize memory accesses on actual dependencies

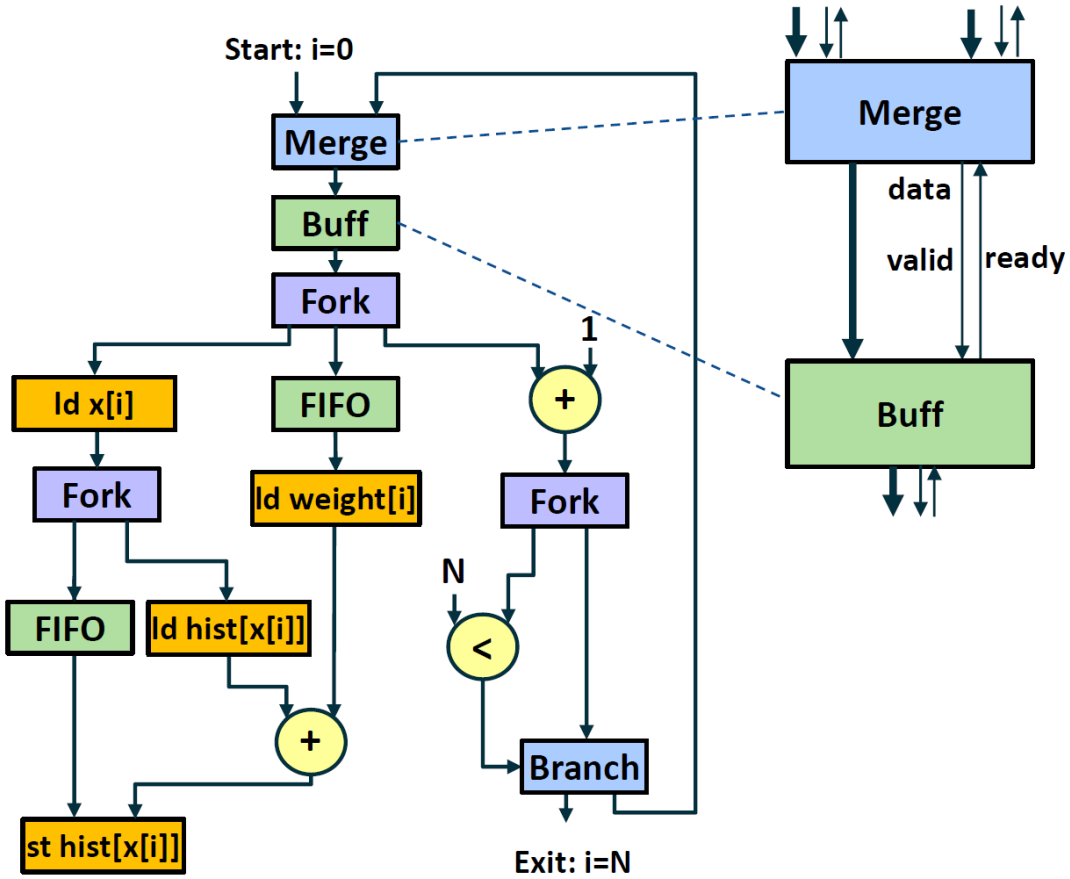


A Different Way to Do HLS

Static scheduling (standard HLS tool): decide at **compile time** when each operation executes

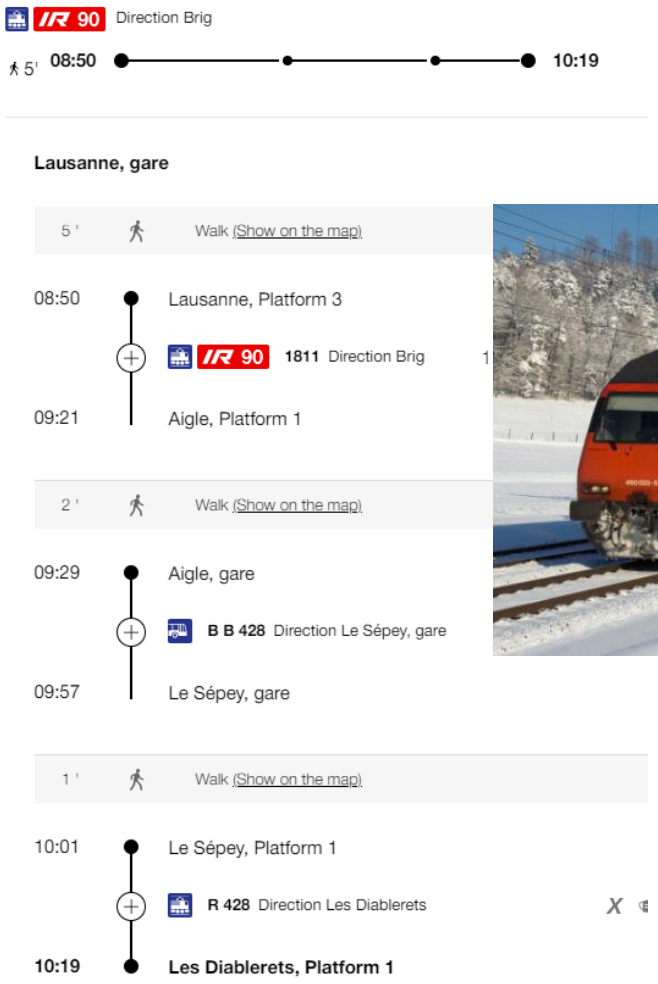


Dynamic scheduling (our HLS approach): decide at **runtime** when each operation executes

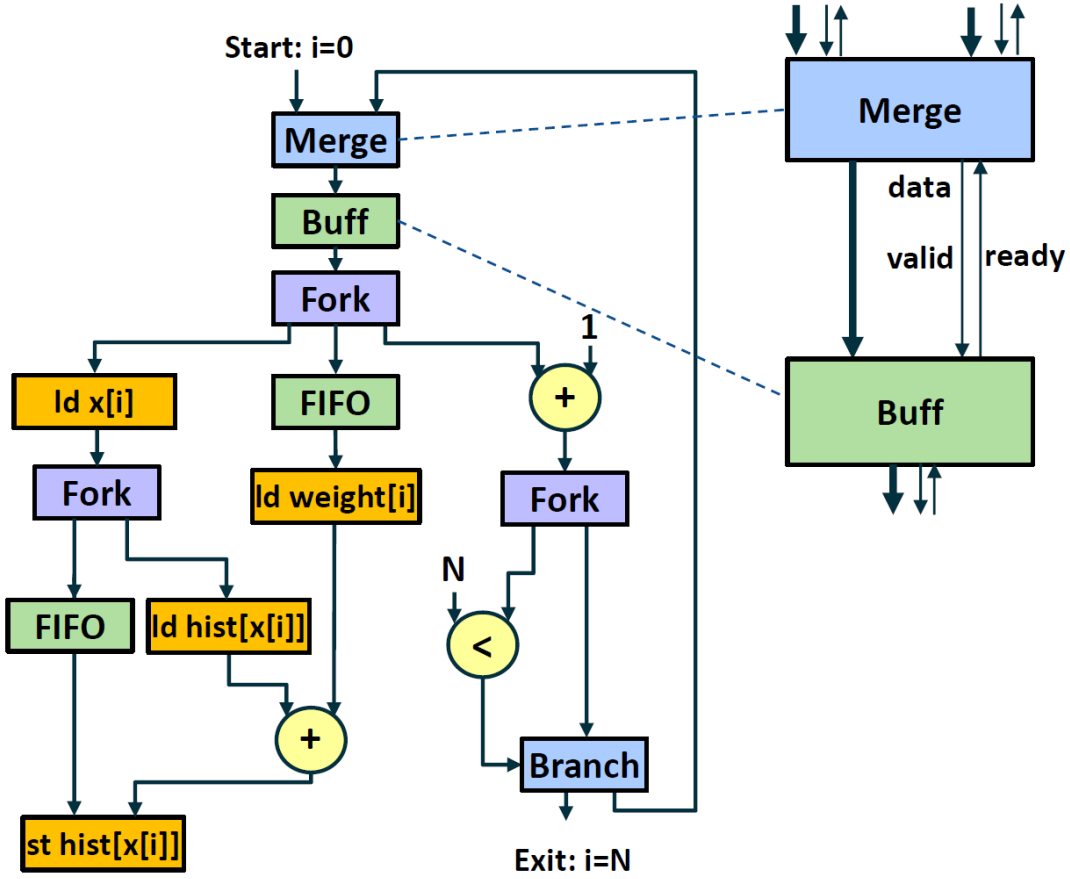


A Different Way to Do HLS

Static scheduling (standard HLS tool): decide at **compile time** when each operation executes



Dynamic scheduling (our HLS approach): decide at **runtime** when each operation executes



A Different Way to Do HLS

Static scheduling (standard HLS tool): decide at **compile time** when each operation executes

IR 90 Direction Brig

5' 08:50 — 10:19

Lausanne, gare

5 ' Walk (Show on the map)

08:50 Lausanne, Platform 3

IR 90 1811 Direction Brig

09:21 Aigle, Platform 1

2 ' Walk (Show on the map)

09:29 Aigle, gare

B B 428 Direction Le Sépey, gare

09:57 Le Sépey, gare

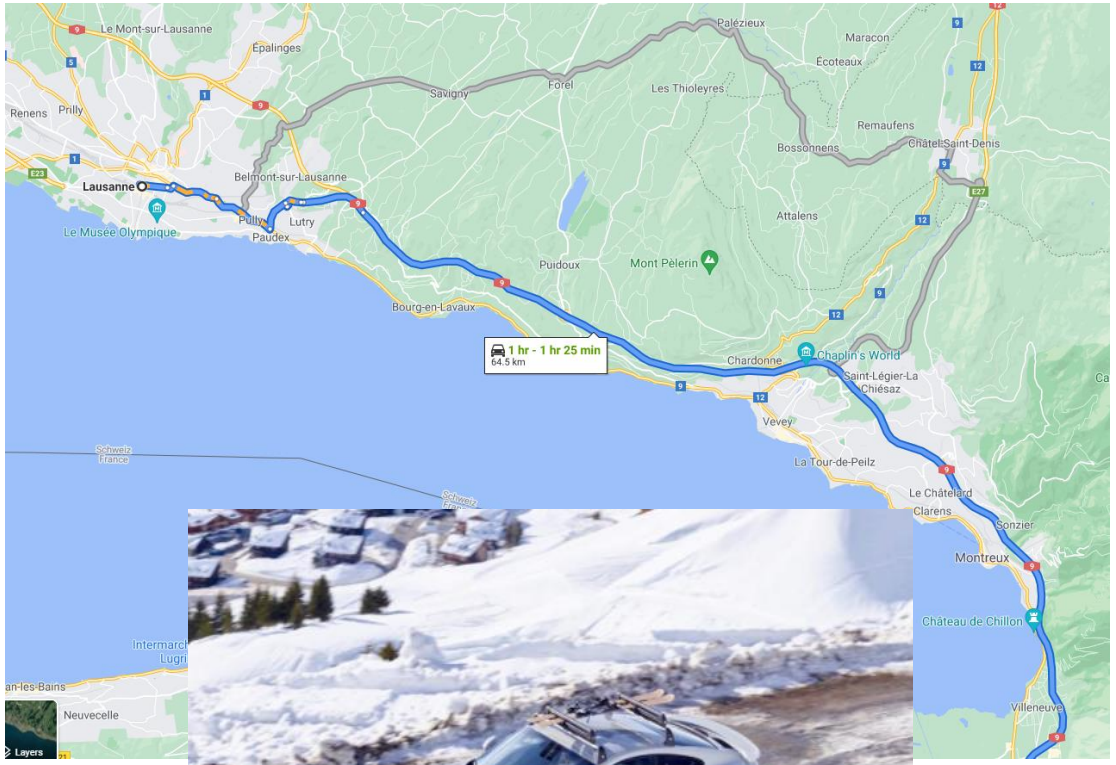
1 ' Walk (Show on the map)

10:01 Le Sépey, Platform 1

R 428 Direction Les Diablerets

10:19 **Les Diablerets, Platform 1**

Dynamic scheduling (our HLS approach): decide at **runtime** when each operation executes



Dynamically Scheduled Circuits

- **Asynchronous circuits**: operators triggered when inputs are available
  - Budiu et al. Dataflow: A complement to superscalar. ISPASS'05.
- Dataflow, latency-insensitive, elastic: the **synchronous** version of it
  - Cortadella et al. Synthesis of synchronous elastic architectures. DAC'06.
  - Carloni et al. Theory of latency-insensitive design. TCAD'01.
  - Jacobson et al. Synchronous interlocked pipelines. ASYNC'02.
  - Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. MEMOCODE'09.

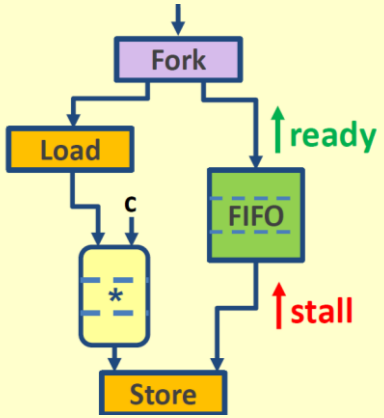
High-level synthesis of dynamically scheduled circuits

HLS of Dynamically Scheduled Circuits

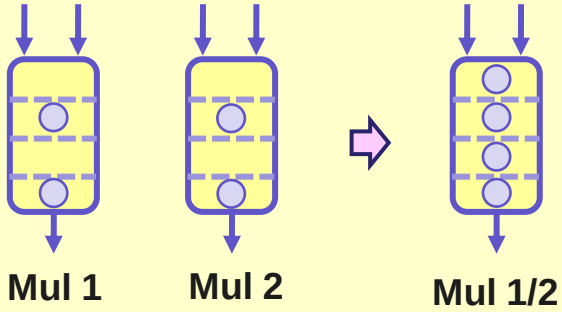
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining



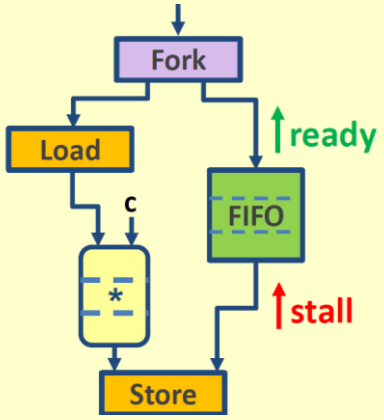
Resource sharing



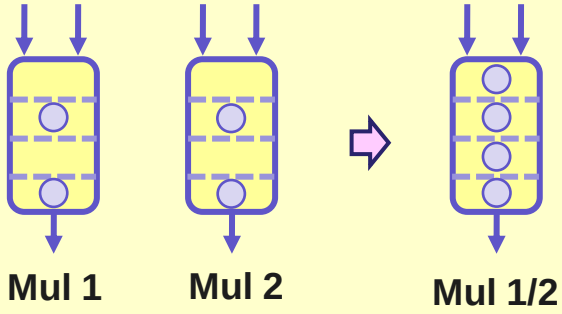
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

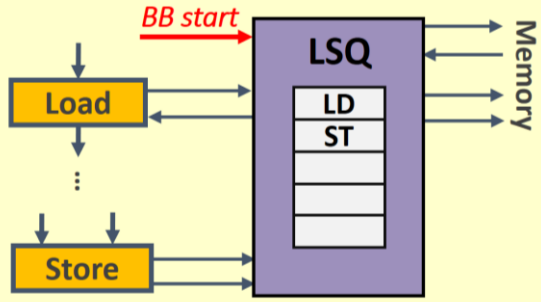


Resource sharing

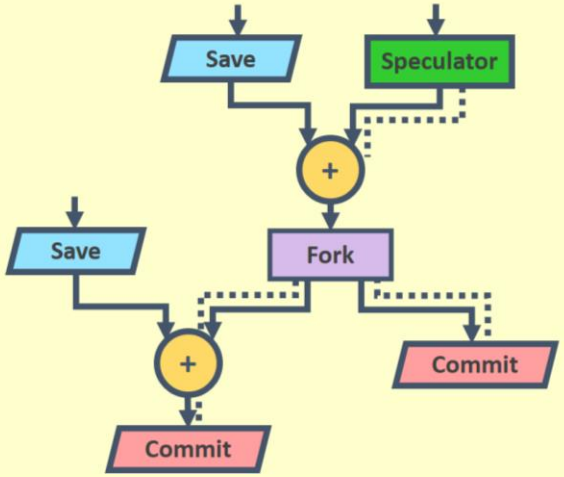


Reaping the benefits of dynamic scheduling

Out-of-order memory

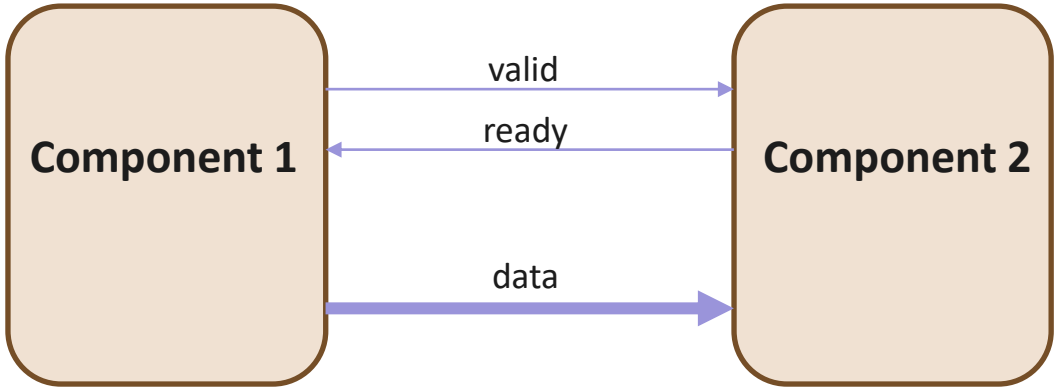


Speculative execution



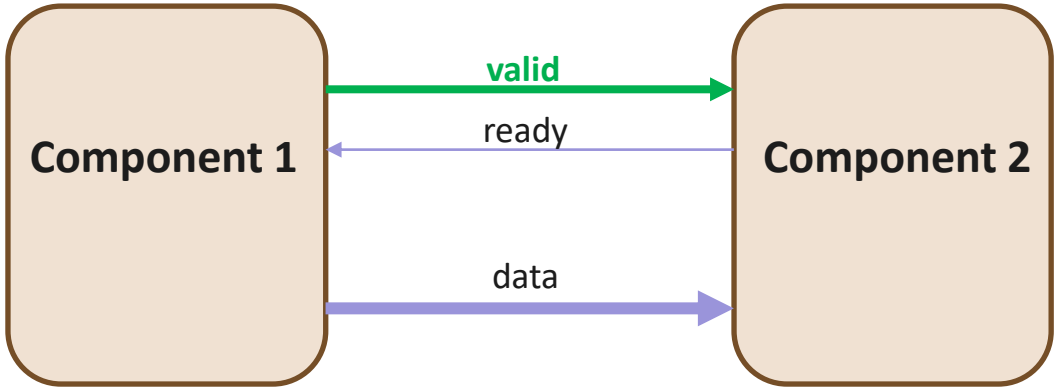
Dataflow Circuits

- We use the **SELF (Synchronous ELastic Flow)** protocol
  - Cortadella et al. Synthesis of synchronous elastic architectures. DAC'06.
- Every component communicates via a pair of handshake signals
- **Make scheduling decisions at runtime**
  - As soon as all conditions for execution are satisfied, an operation starts



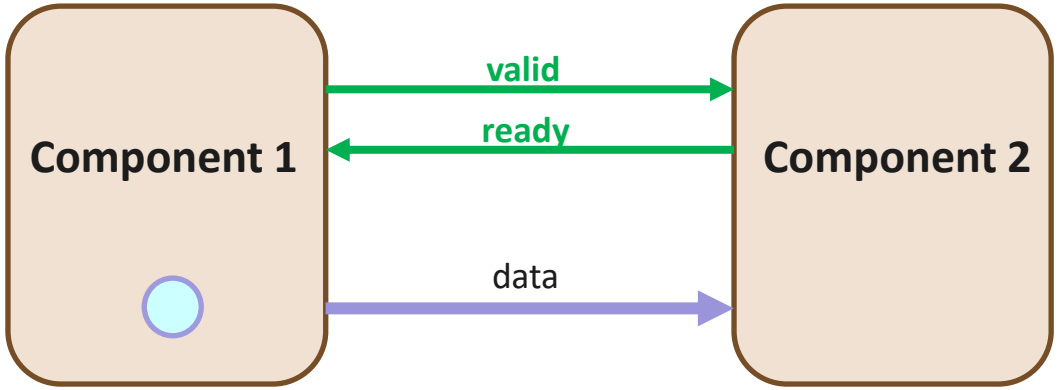
Dataflow Circuits

- We use the **SELF (Synchronous ELastic Flow)** protocol
  - Cortadella et al. Synthesis of synchronous elastic architectures. DAC'06.
- Every component communicates via a pair of handshake signals
- **Make scheduling decisions at runtime**
  - As soon as all conditions for execution are satisfied, an operation starts



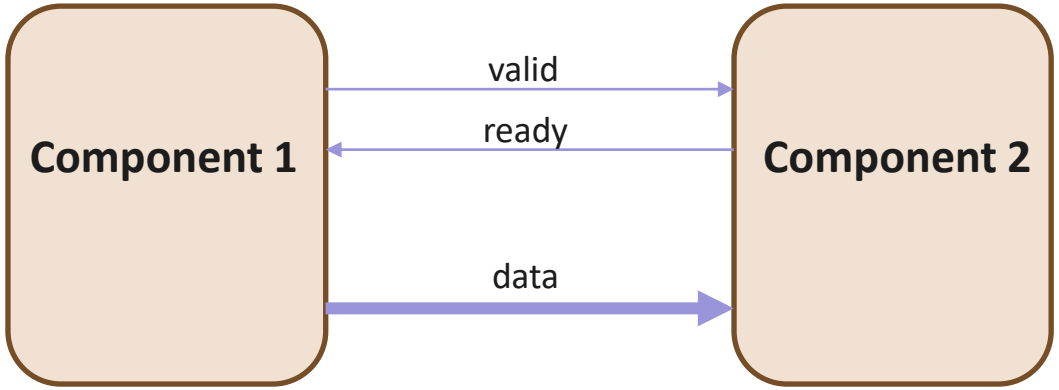
Dataflow Circuits

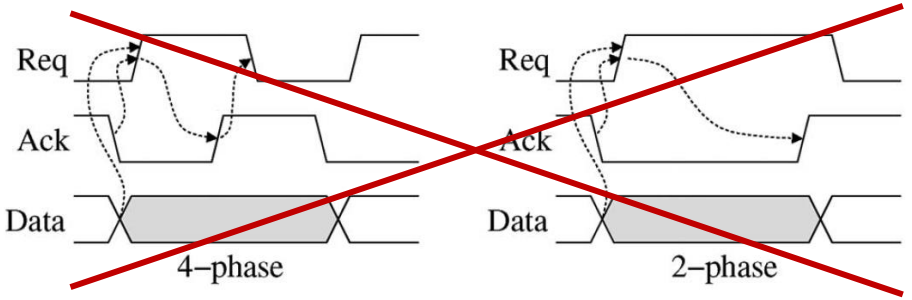
- We use the **SELF (Synchronous ELastic Flow)** protocol
  - Cortadella et al. Synthesis of synchronous elastic architectures. DAC'06.
- Every component communicates via a pair of handshake signals
- **Make scheduling decisions at runtime**
  - As soon as all conditions for execution are satisfied, an operation starts

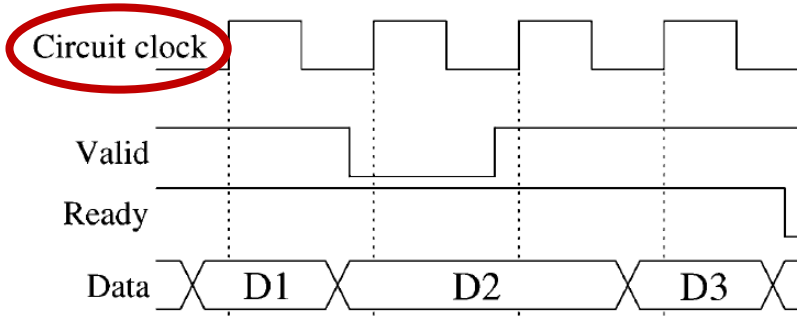


Dataflow Circuits

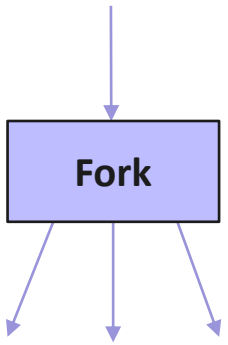
- We use the **SELF (Synchronous ELastic Flow)** protocol
  - Cortadella et al. Synthesis of synchronous elastic architectures. DAC'06.
- Every component communicates via a pair of handshake signals
- **Make scheduling decisions at runtime**
  - As soon as all conditions for execution are satisfied, an operation starts



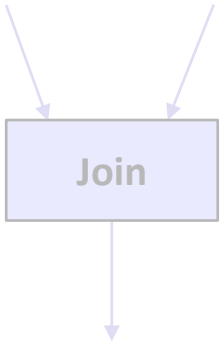




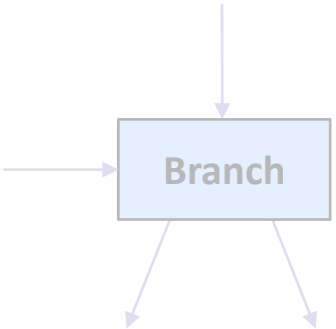
Dataflow Components



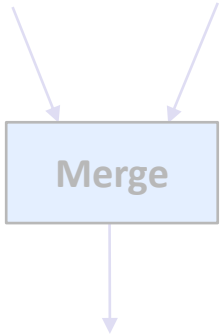
Fork



Join

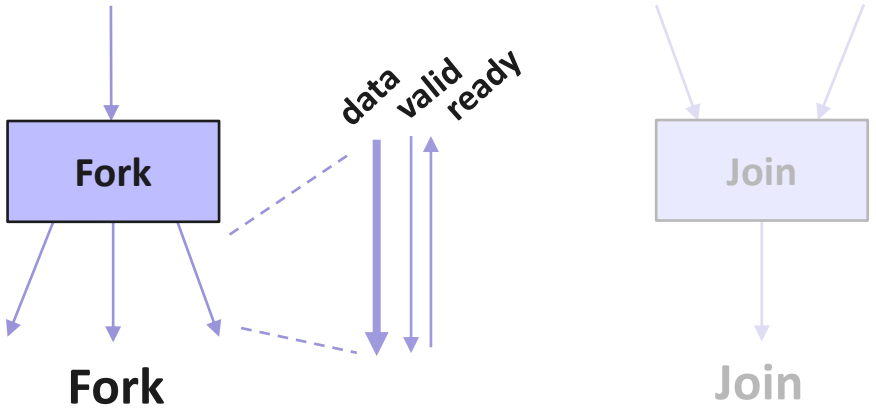


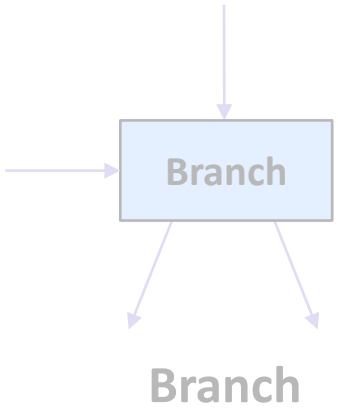
Branch

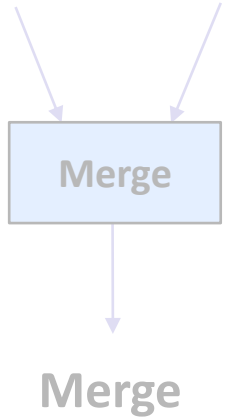


Merge

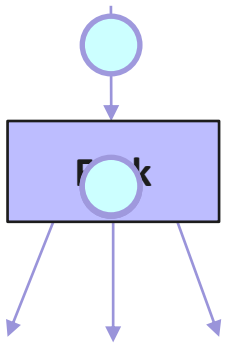
Dataflow Components



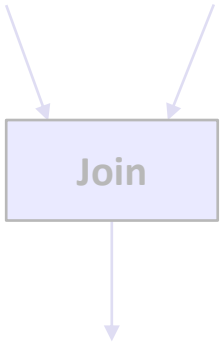




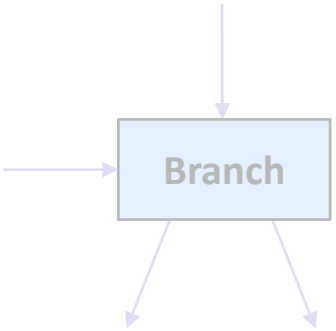
Dataflow Components



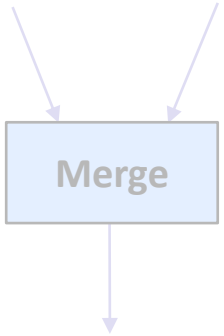
Fork



Join

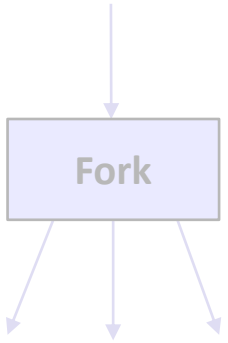


Branch

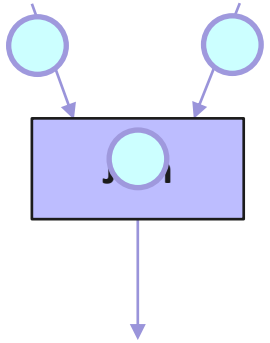


Merge

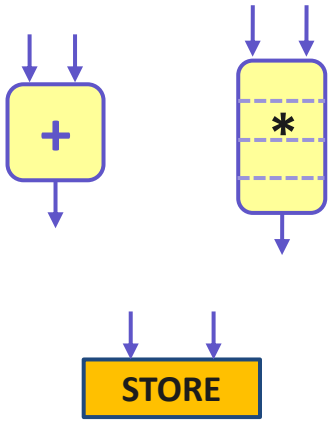
Dataflow Components

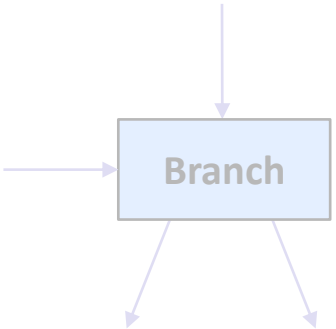


Fork

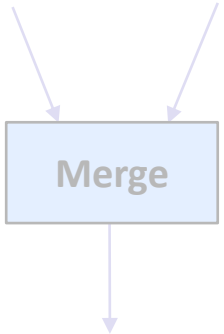


Join



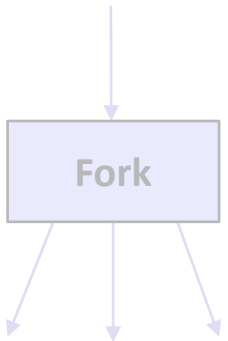


Branch

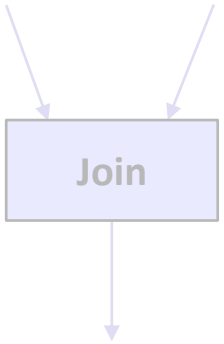


Merge

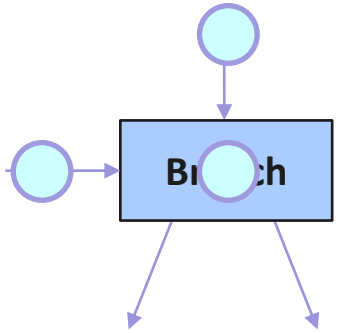
Dataflow Components



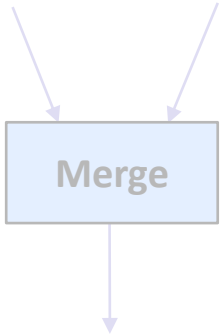
Fork



Join

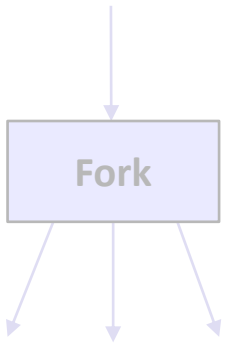


Branch

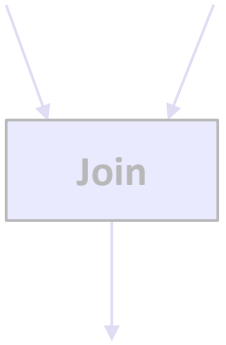


Merge

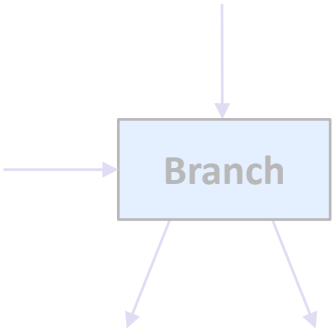
Dataflow Components



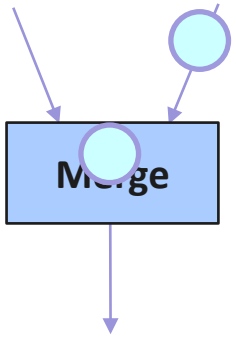
Fork



Join

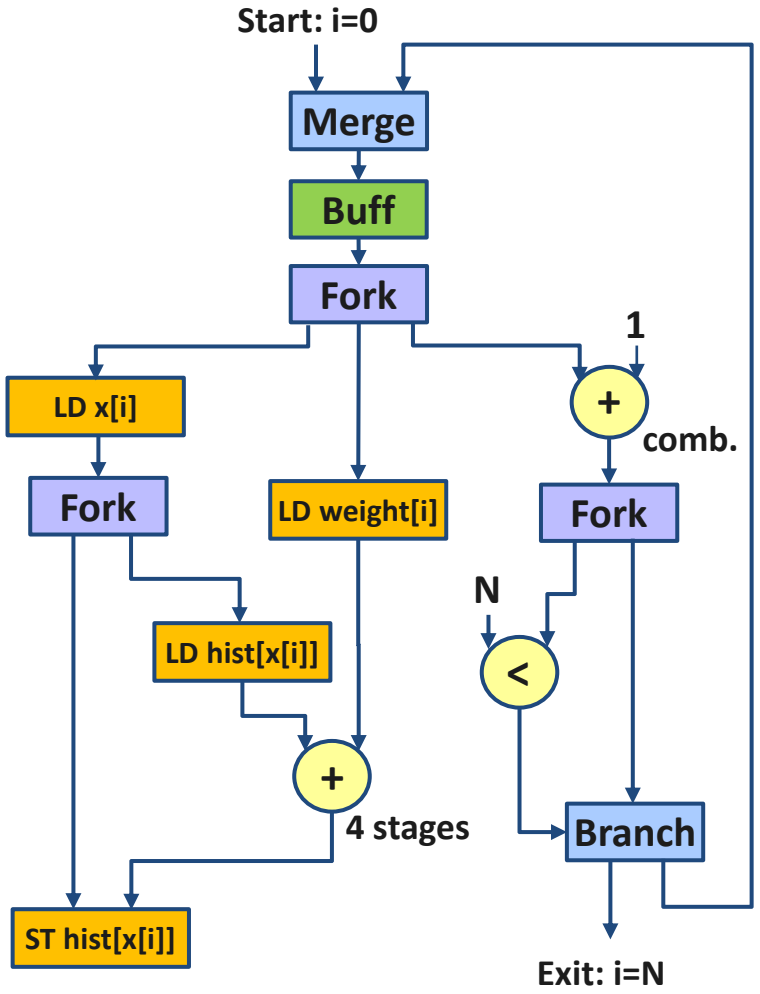


Branch



Merge

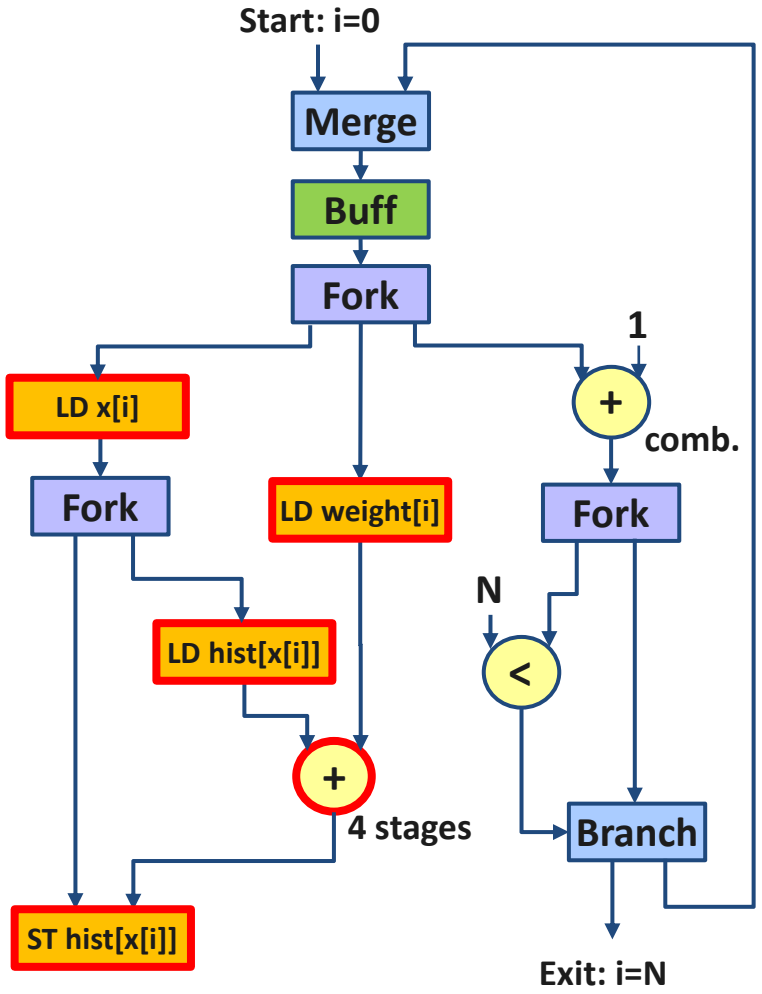
From Program to Dataflow Circuit




```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

Josipović, Ghosal, and Ienne. Dynamically Scheduled High-Level Synthesis. FPGA 2018 **Best Paper Award Nominee**
Josipović, Brisk, and Ienne. From C to Elastic Circuits. Asilomar 2017

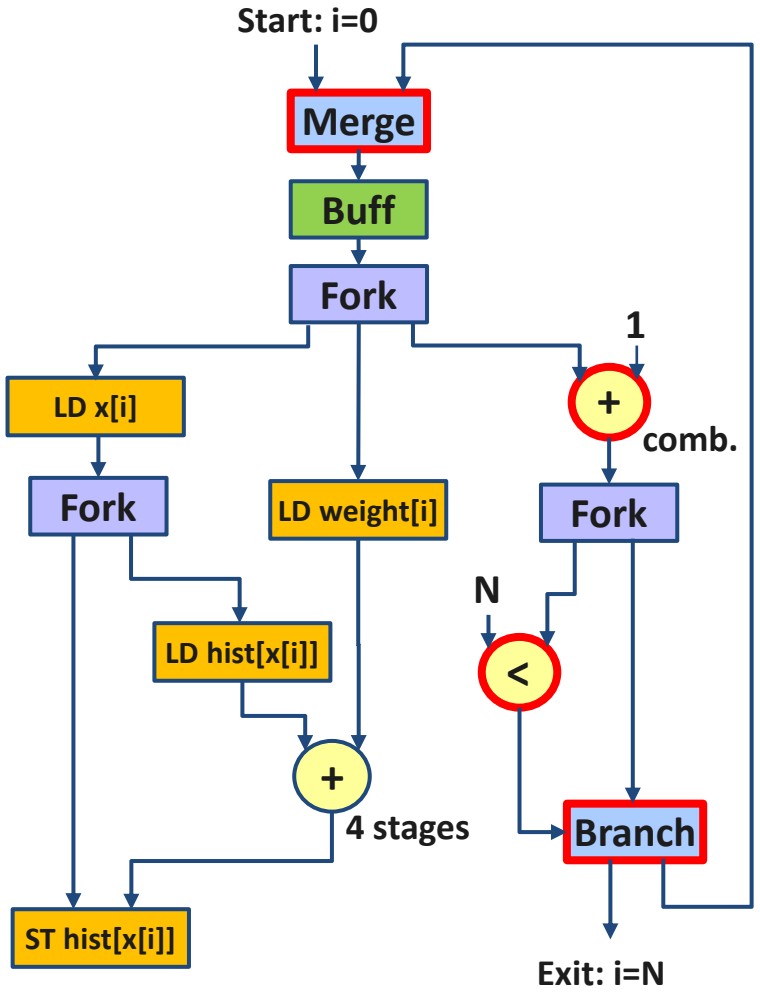
From Program to Dataflow Circuit




```
for (i=0; i<N; i++) {
  hist[x[i]] = hist[x[i]] + weight[i];
}
```

Josipović, Ghosal, and Ienne. Dynamically Scheduled High-Level Synthesis. FPGA 2018 **Best Paper Award Nominee**
Josipović, Brisk, and Ienne. From C to Elastic Circuits. Asilomar 2017

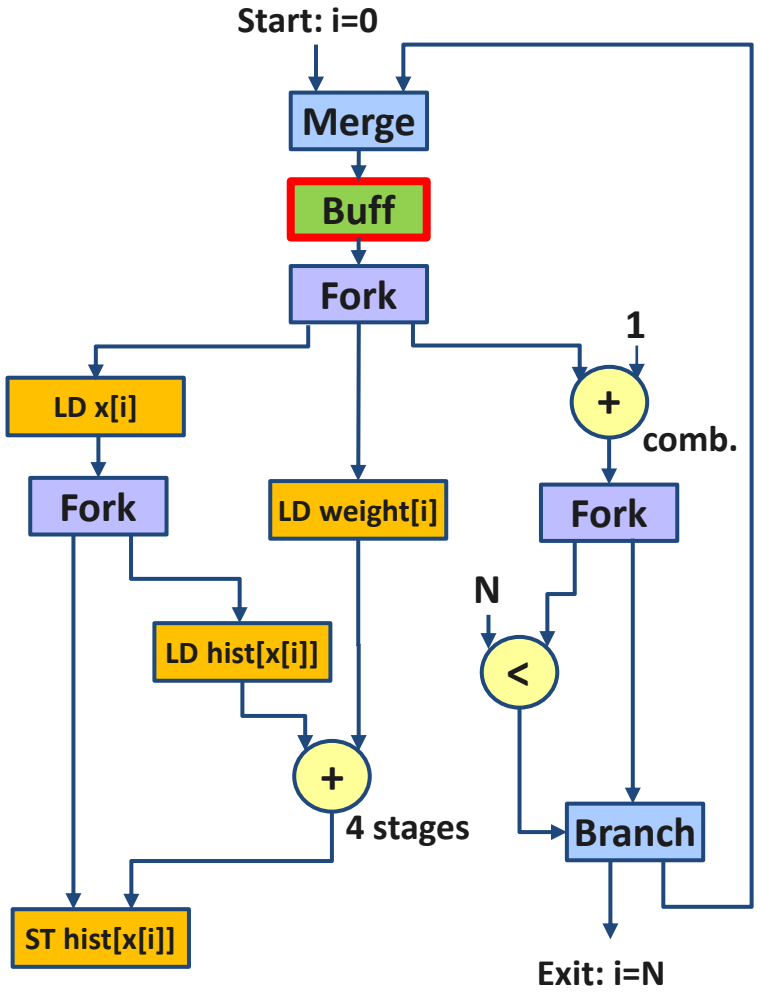
From Program to Dataflow Circuit




```
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
```

Josipović, Ghosal, and Ienne. Dynamically Scheduled High-Level Synthesis. FPGA 2018 **Best Paper Award Nominee**
Josipović, Brisk, and Ienne. From C to Elastic Circuits. Asilomar 2017

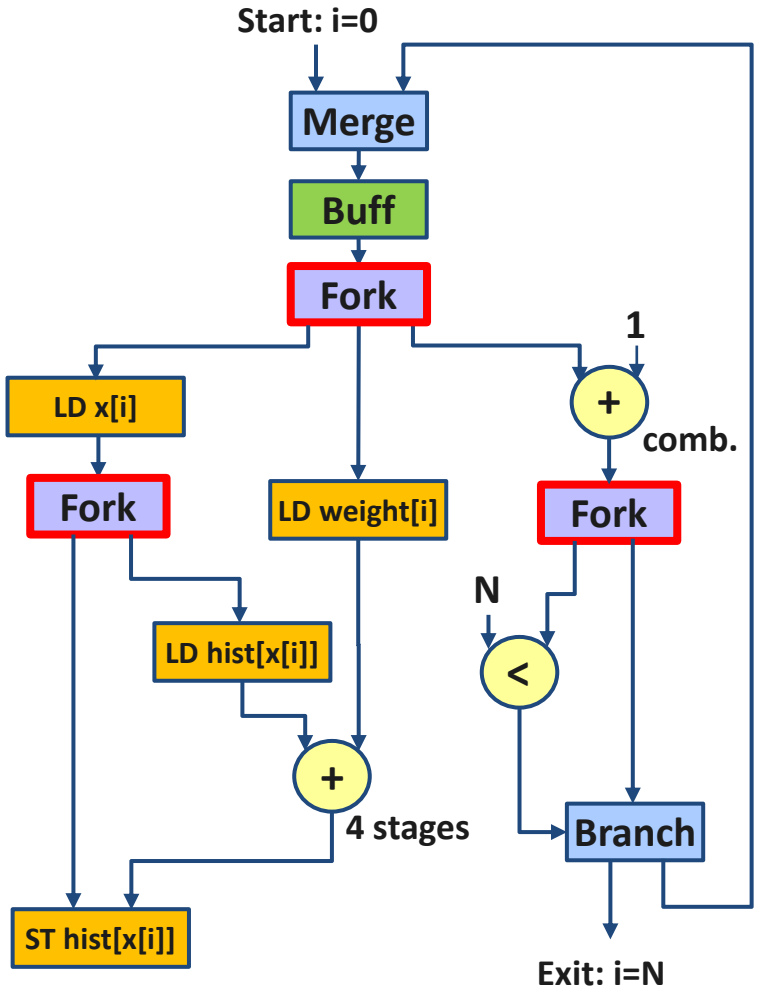
From Program to Dataflow Circuit




```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

Josipović, Ghosal, and Ienne. Dynamically Scheduled High-Level Synthesis. FPGA 2018 **Best Paper Award Nominee**
Josipović, Brisk, and Ienne. From C to Elastic Circuits. Asilomar 2017

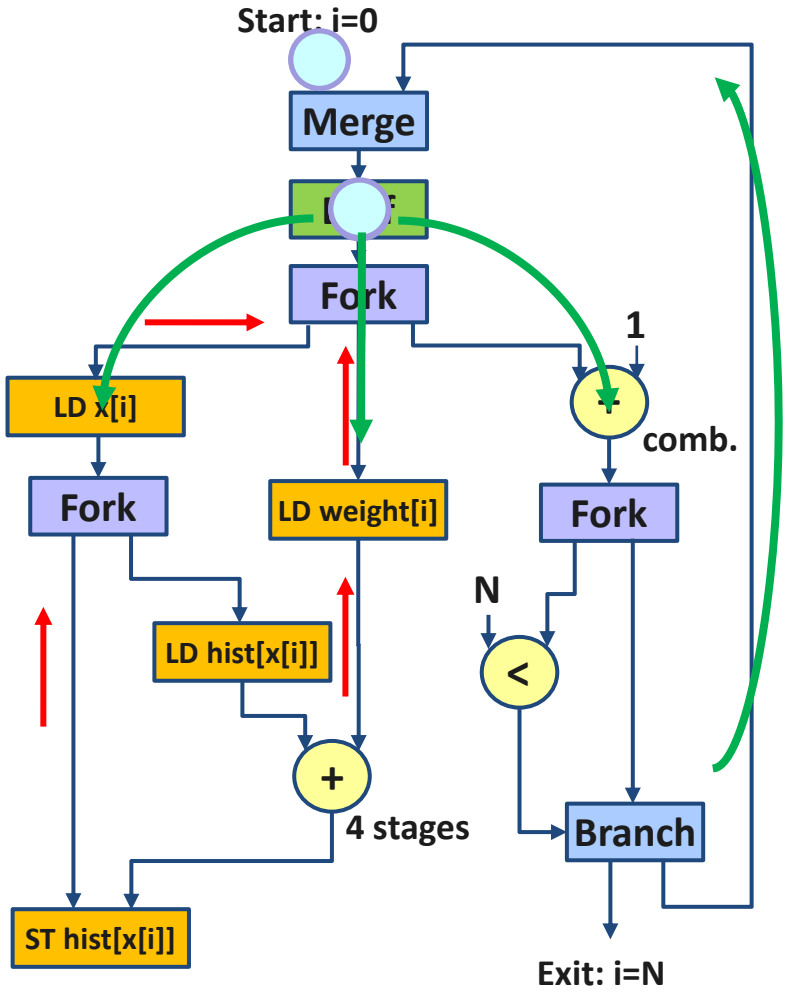
From Program to Dataflow Circuit




```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

Josipović, Ghosal, and Ienne. Dynamically Scheduled High-Level Synthesis. FPGA 2018 **Best Paper Award Nominee**
Josipović, Brisk, and Ienne. From C to Elastic Circuits. Asilomar 2017

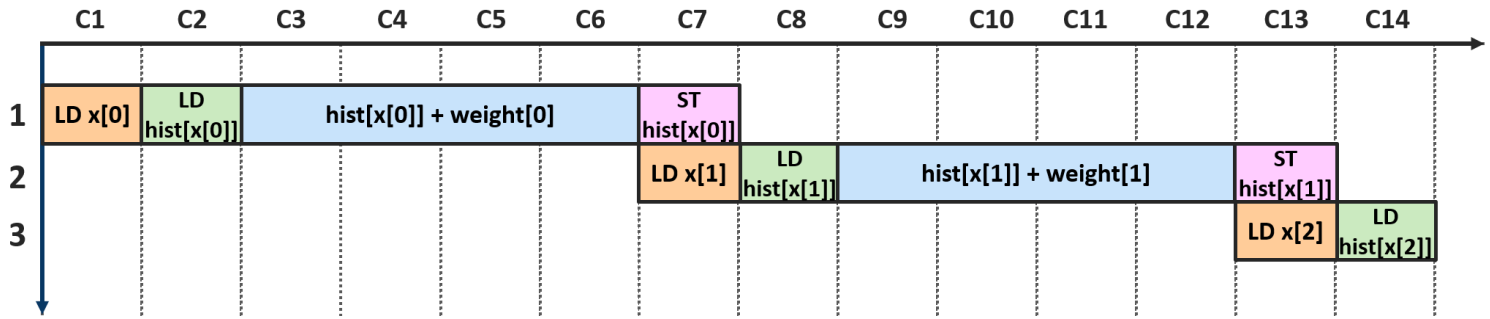
From Program to Dataflow Circuit



Single token on cycle, in-order tokens in noncyclic paths

Josipović, Ghosal, and Ienne. Dynamically Scheduled High-Level Synthesis. FPGA 2018 **Best Paper Award Nominee**
Josipović, Brisk, and Ienne. From C to Elastic Circuits. Asilomar 2017

From Program to Dataflow Circuit

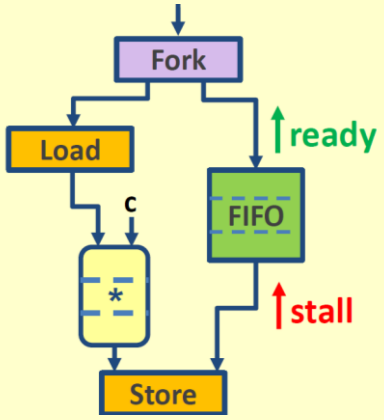


Backpressure from slow paths prevents pipelining

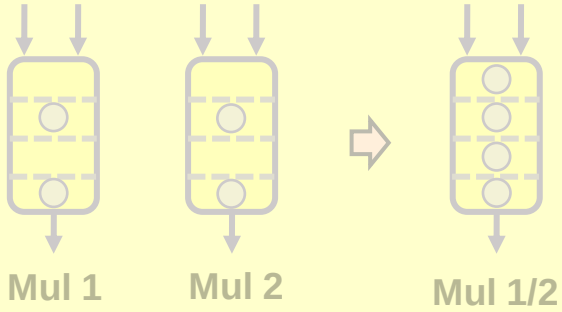
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

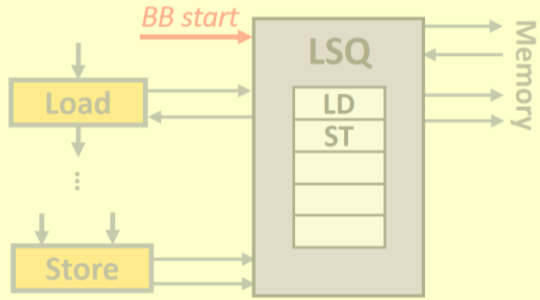


Resource sharing

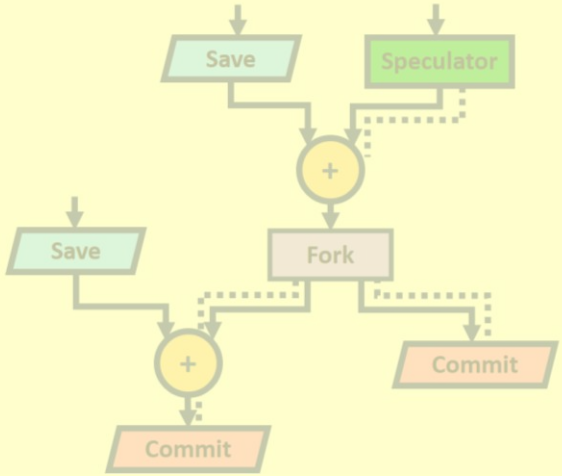


Reaping the benefits of dynamic scheduling

Out-of-order memory

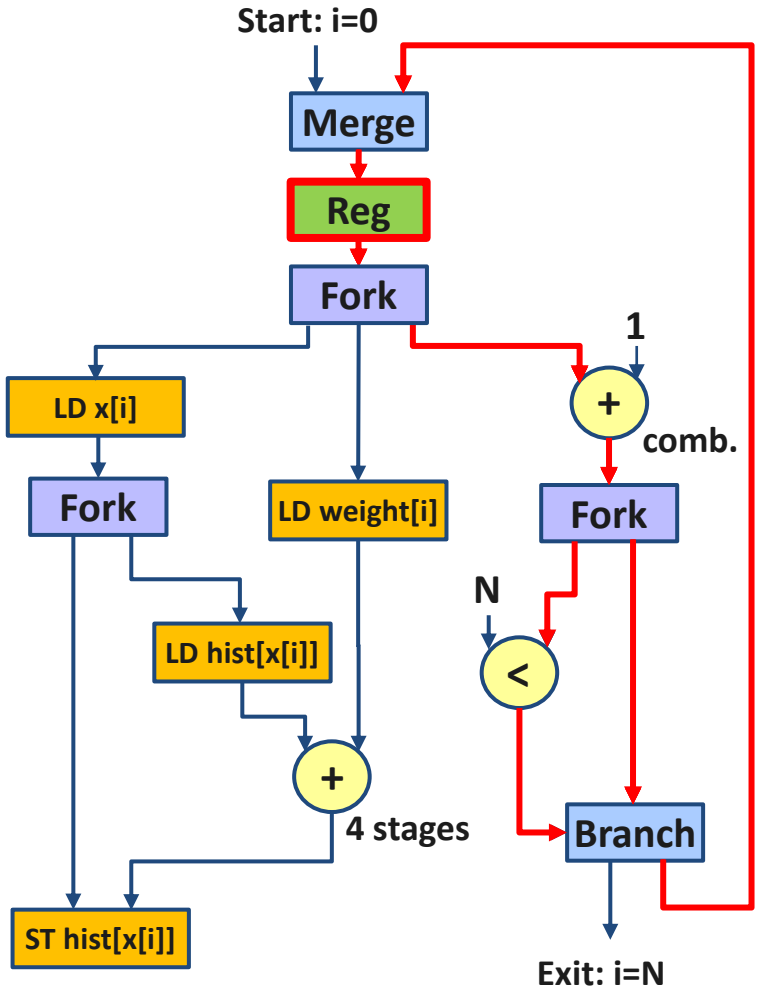


Speculative execution



Inserting Buffers

```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

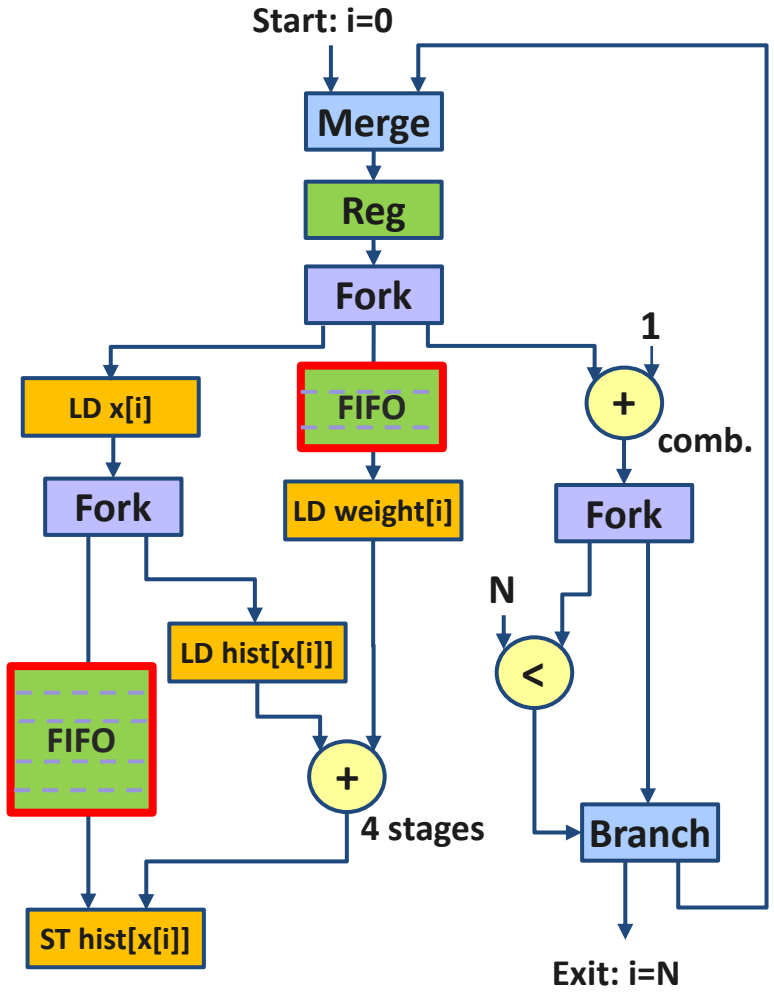


Buffers as registers to break combinational paths

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 **Best paper award**
Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022

Inserting Buffers

```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

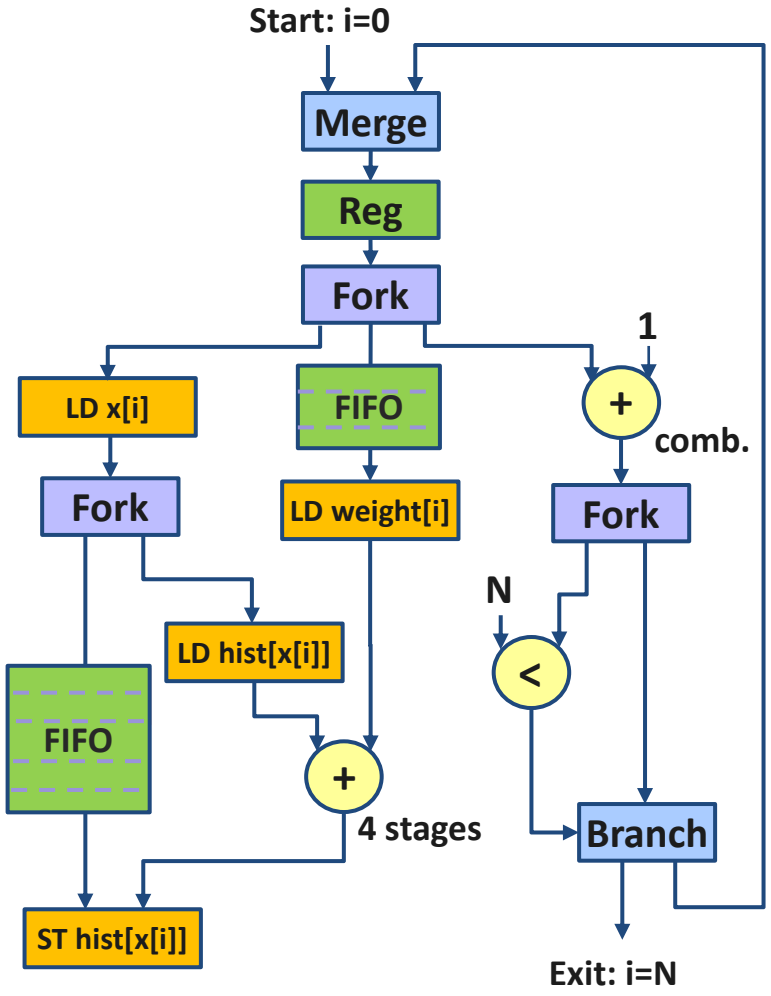


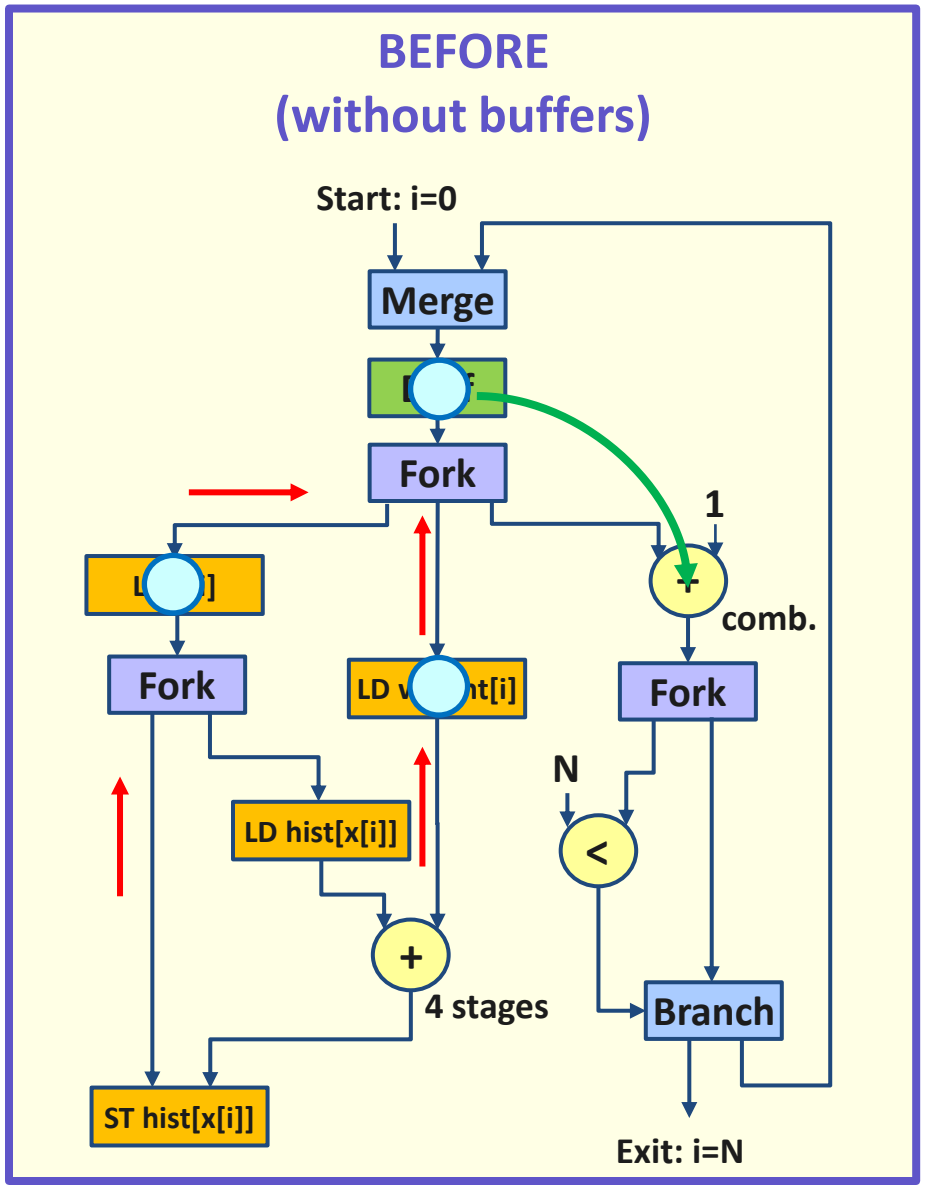
Buffers as FIFOs to regulate throughput

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 **Best paper award**
Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022

Inserting Buffers

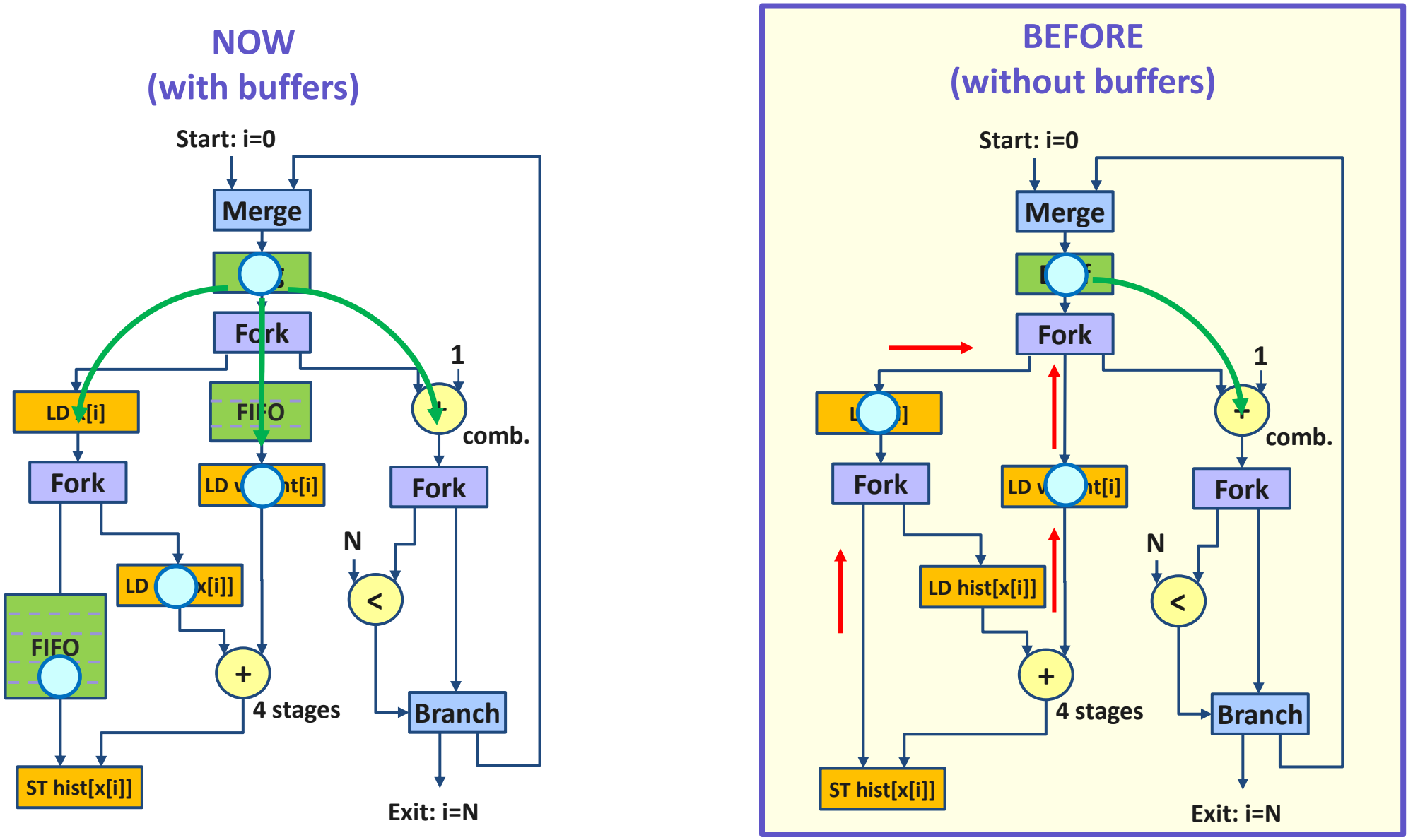
```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```





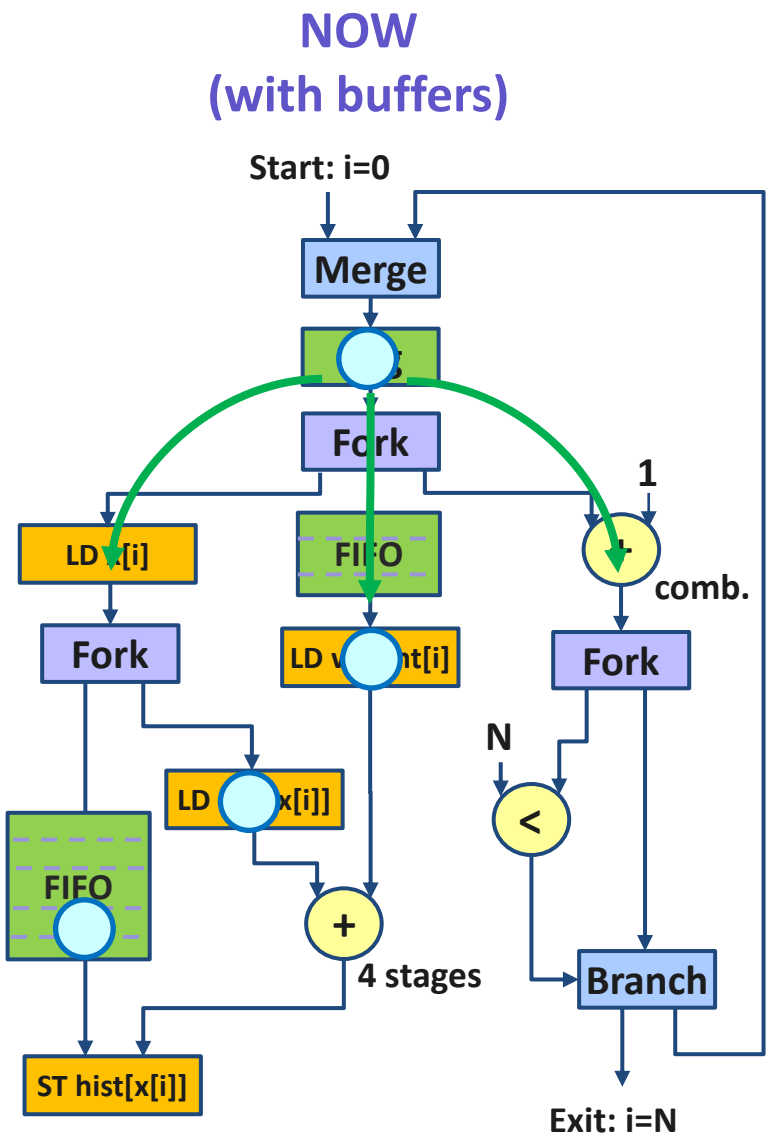
Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 **Best paper award**
Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022

Inserting Buffers



Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 **Best paper award**
Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022

Inserting Buffers



Mixed integer linear programming (MILP) model based on **Petri net theory**

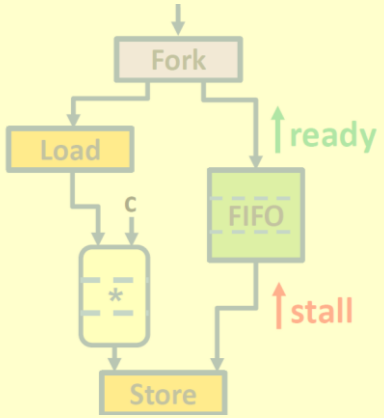
- Analyze token flow through the circuit
- Determine **buffer placement and sizing**
- **Maximize throughput** for a target clock period

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 **Best paper award**
Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022

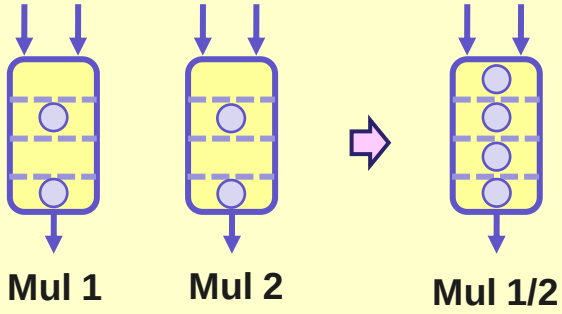
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

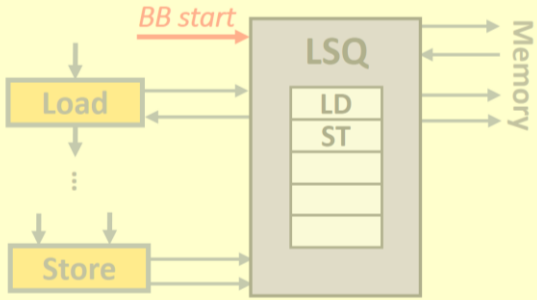


Resource sharing

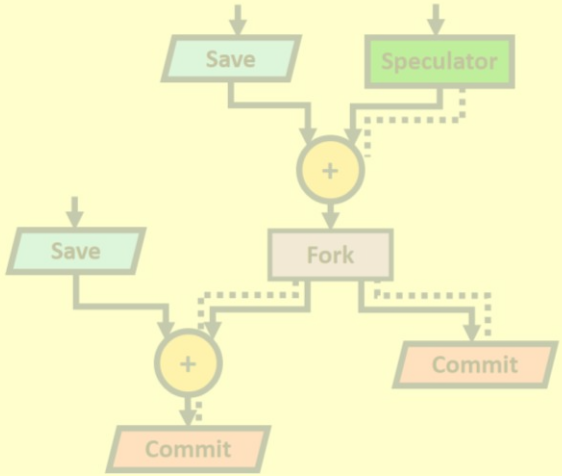


Reaping the benefits of dynamic scheduling

Out-of-order memory



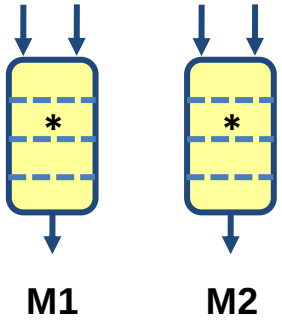
Speculative execution



Saving Resources through Sharing

- Static HLS: share units between operations which execute in **different clock cycles**

```
for (i = 0; i < N; i++) {
    a[i] = a[i]*x;
    b[i] = b[i]*y;
}
```

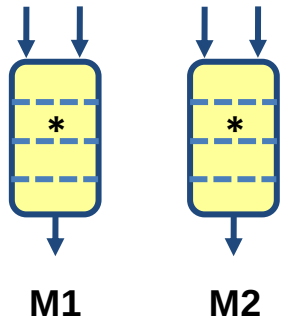


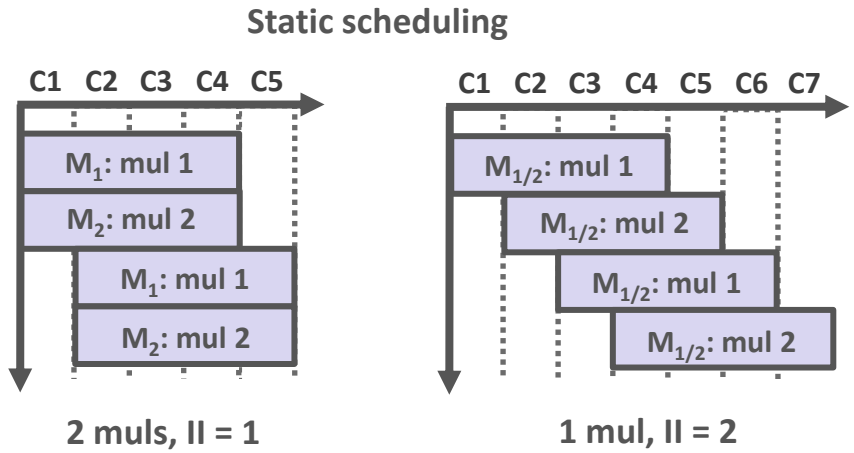
Josipović, Marmet, Guerrieri, and Ienne. Resource Sharing in Dataflow Circuits. FCCM 2022. **Best Paper Award Nominee**

Saving Resources through Sharing

- Static HLS: share units between operations which execute in **different clock cycles**

```
for (i = 0; i < N; i++) {
    a[i] = a[i]*x;
    b[i] = b[i]*y;
}
```



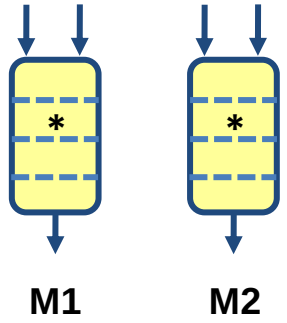


Josipović, Marmet, Guerrieri, and Ienne. Resource Sharing in Dataflow Circuits. FCCM 2022. **Best Paper Award Nominee**

Saving Resources through Sharing

- Static HLS: share units between operations which execute in **different clock cycles**
- Dynamic HLS: share units based on their **average utilization** with tokens

```
for (i = 0; i < N; i++) {
    a[i] = a[i]*x;
    b[i] = b[i]*y;
}
```

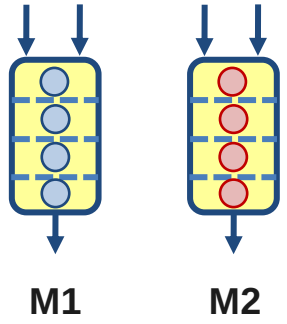


Josipović, Marmet, Guerrieri, and Ienne. Resource Sharing in Dataflow Circuits. FCCM 2022. **Best Paper Award Nominee**

Saving Resources through Sharing

- Static HLS: share units between operations which execute in **different clock cycles**
- Dynamic HLS: share units based on their **average utilization** with tokens

```
for (i = 0; i < N; i++) {
    a[i] = a[i]*x;
    b[i] = b[i]*y;
}
```



Units fully utilized
(high throughput, II = 1)

Sharing not possible without damaging throughput

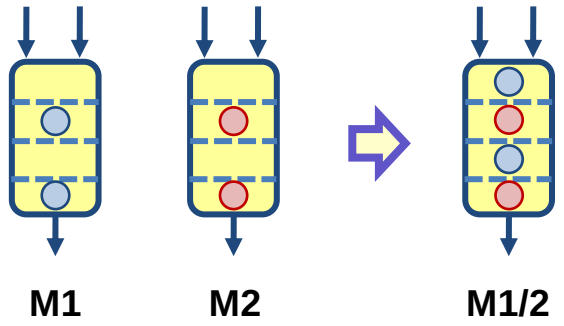
Use throughput information to decide what to share

Josipović, Marmet, Guerrieri, and Ienne. Resource Sharing in Dataflow Circuits. FCCM 2022. **Best Paper Award Nominee**

Saving Resources through Sharing

- Static HLS: share units between operations which execute in **different clock cycles**
- Dynamic HLS: share units based on their **average utilization** with tokens

```
for (i = 0; i < N; i++) {
    a[i] = a[i]*x;
    b[i] = b[i]*y;
}
```



Sharing possible without damaging throughput

Units underutilized (low throughput, II = 2)

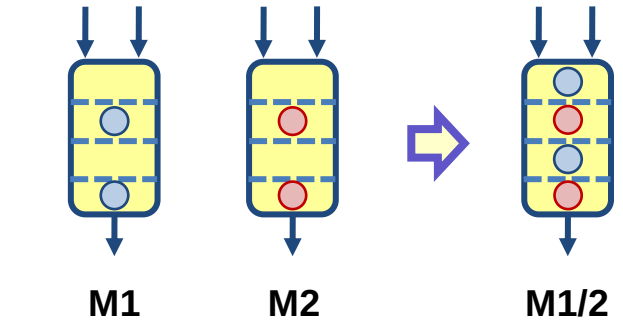
Use throughput information to decide what to share

Josipović, Marmet, Guerrieri, and Ienne. Resource Sharing in Dataflow Circuits. FCCM 2022. **Best Paper Award Nominee**

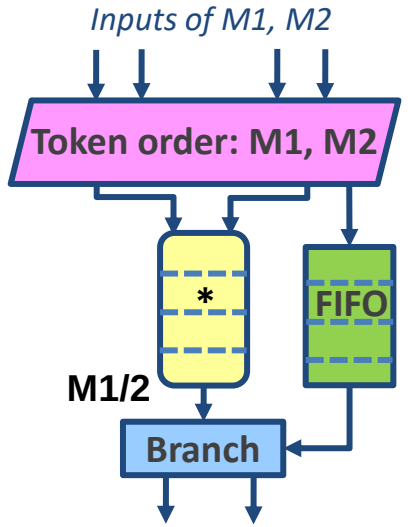
Saving Resources through Sharing

- Static HLS: share units between operations which execute in **different clock cycles**
- Dynamic HLS: share units based on their **average utilization** with tokens

```
for (i = 0; i < N; i++) {
    a[i] = a[i]*x;
    b[i] = b[i]*y;
}
```



Units underutilized
(low throughput, II = 2)

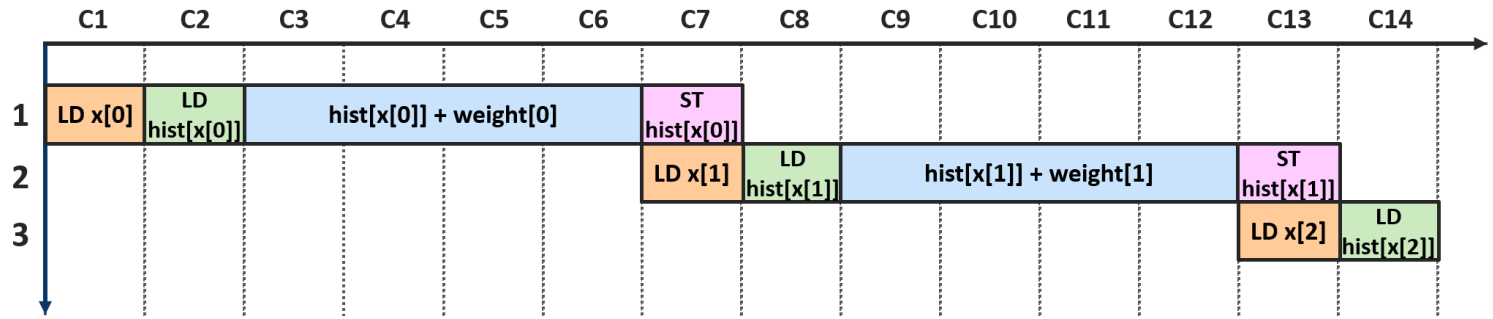


Sharing mechanism for deadlock-free execution

Josipović, Marmet, Guerrieri, and Ienne. Resource Sharing in Dataflow Circuits. FCCM 2022. **Best Paper Award Nominee**

Inserting Buffers

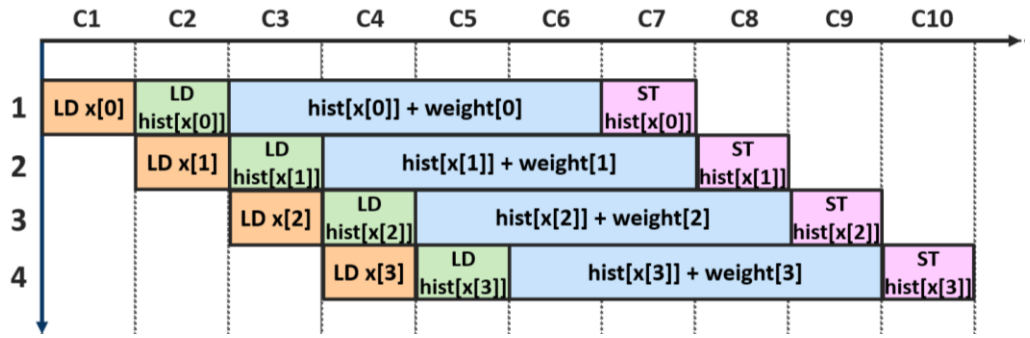
```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```



Backpressure from slow paths prevents pipelining

Inserting Buffers

```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```



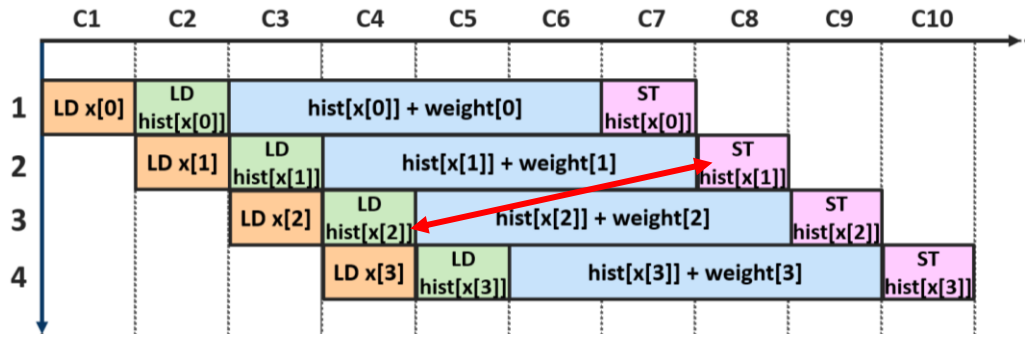
Buffers for high throughput

Inserting Buffers

```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

```
1: x[0]=5 → ld hist[5]; st hist[5];
2: x[1]=4 → ld hist[4]; st hist[4];
3: x[2]=4 → ld hist[4]; st hist[4];
```

RAW dependency



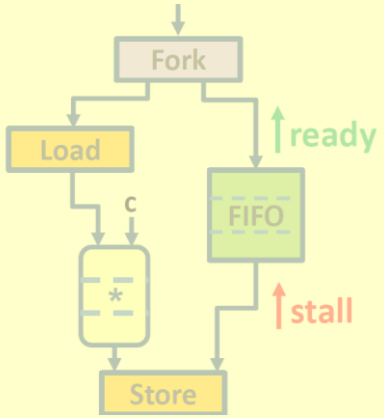
RAW dependency not honored!

What about memory?

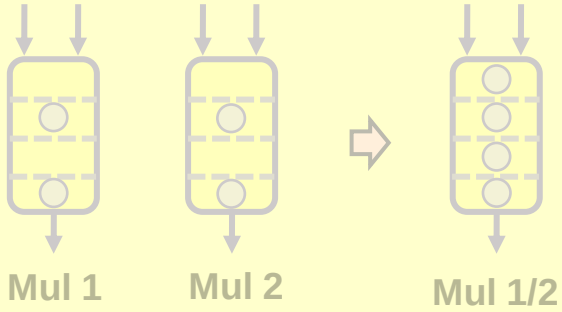
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

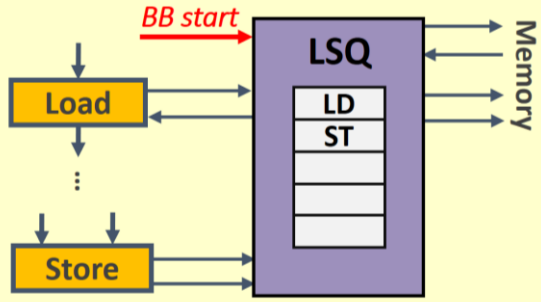


Resource sharing

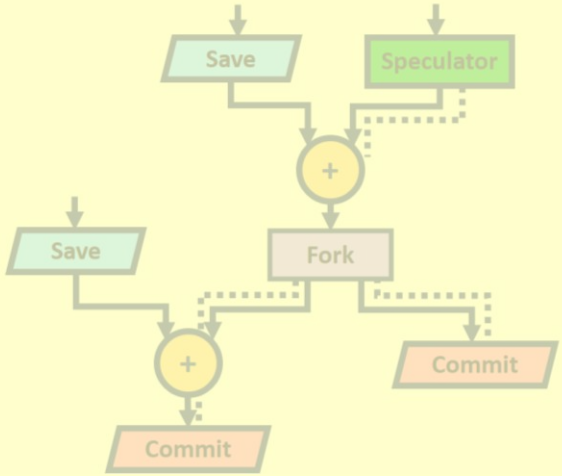


Reaping the benefits of dynamic scheduling

Out-of-order memory

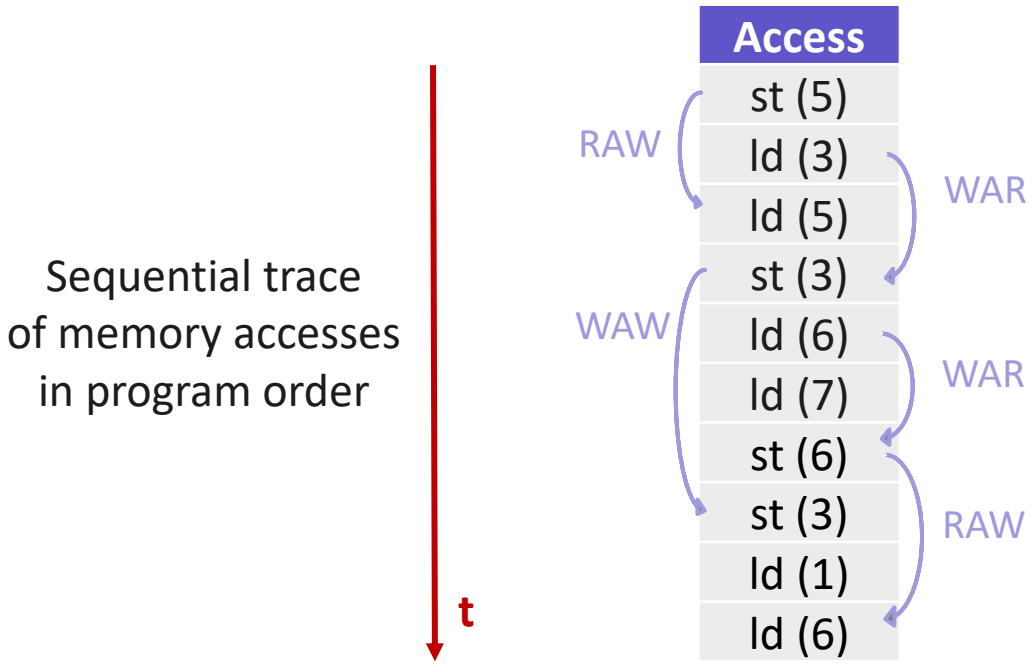


Speculative execution



The Ordering Problem

- A dataflow circuit **may reorder memory accesses** in (almost) any way
- We need to keep RAWs, WAWs, and WARs in the **original program order**



RAW = Read after write
st (*n*) → ld (*n*)

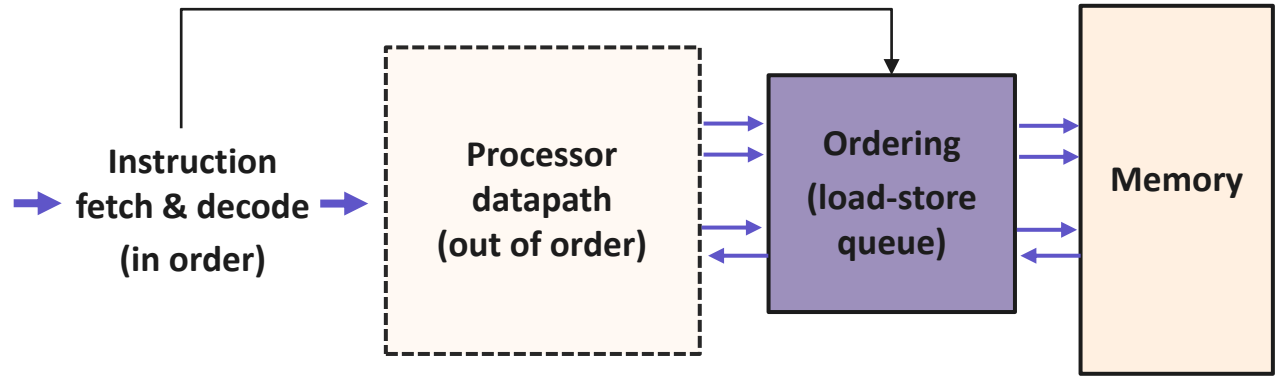
WAW = Write after write
st (*n*) → st (*n*)

WAR = Write after read
ld (*n*) → st (*n*)

We Need a Load-Store Queue (LSQ)!

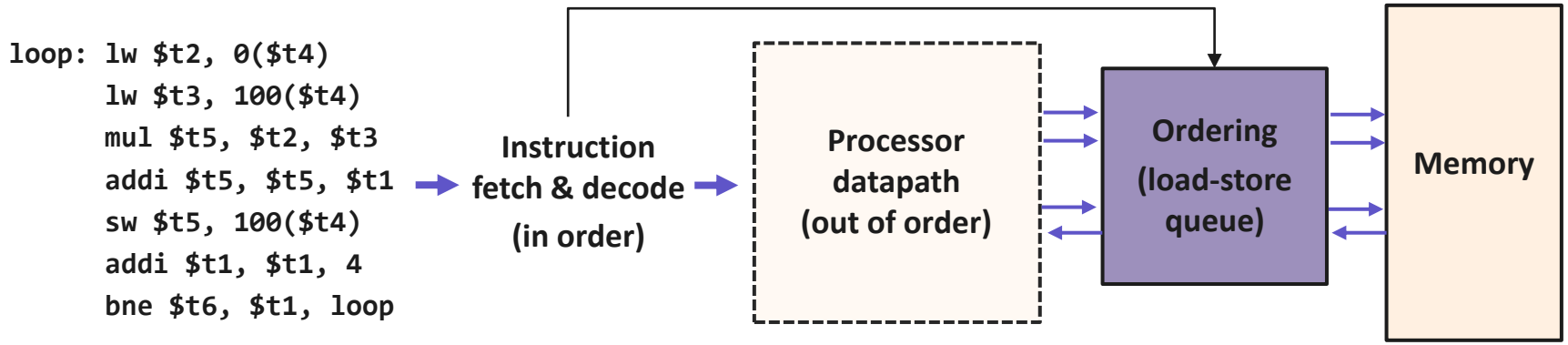
- Processor LSQs keep dependent memory accesses **in the original program order**

```
loop: lw $t2, 0($t4)
      lw $t3, 100($t4)
      mul $t5, $t2, $t3
      addi $t5, $t5, $t1
      sw $t5, 100($t4)
      addi $t1, $t1, 4
      bne $t6, $t1, loop
```

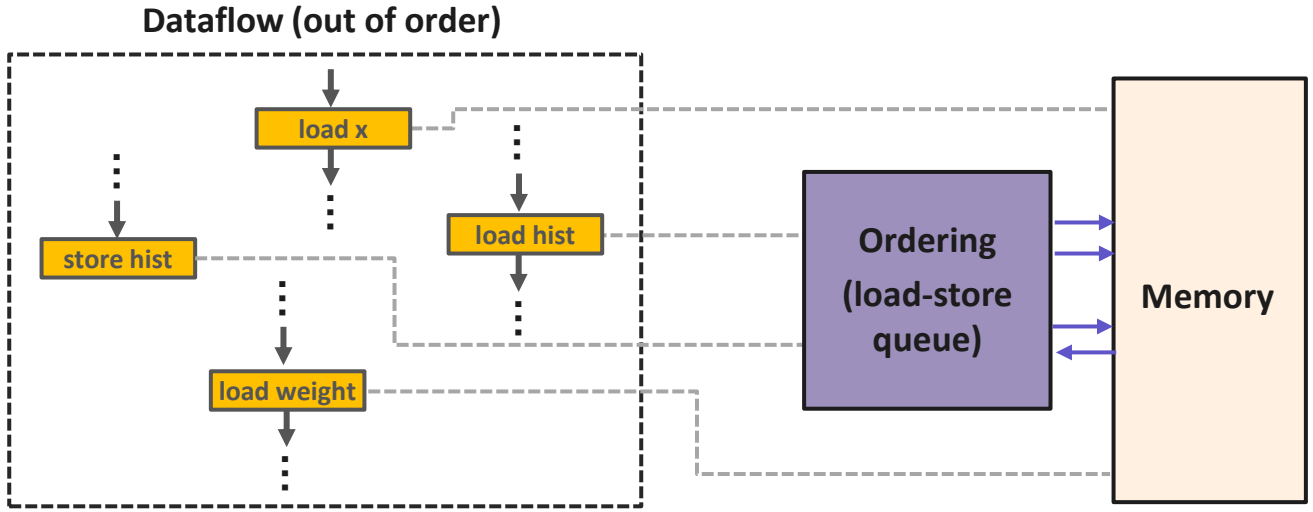


We Need a Load-Store Queue (LSQ)!

- Processor LSQs keep dependent memory accesses **in the original program order**



- **Application-specific LSQs** for dataflow circuits

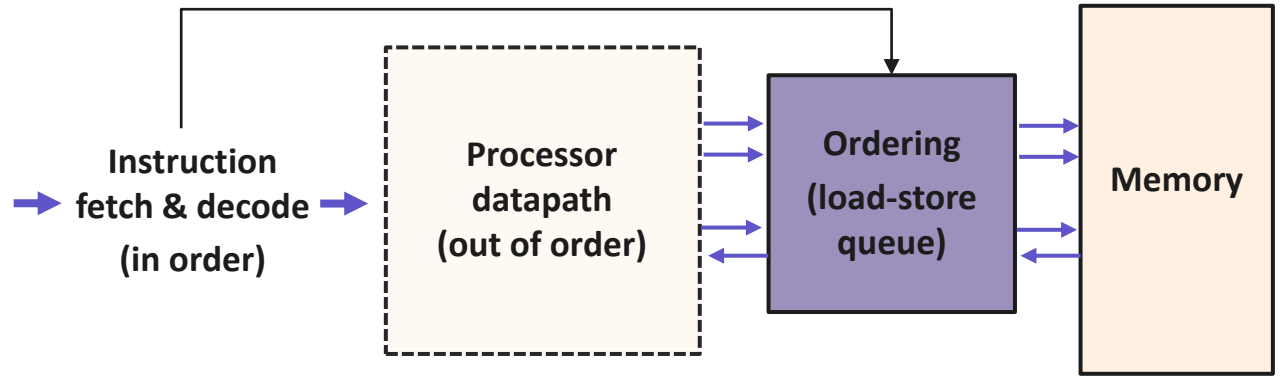


Josipović, Brisk, and Ienne. An Out-of-Order Load-Store Queue for Spatial Computing. CASES 2017 **Best Paper Award Nominee**
Liu, Rizzi, and Josipović. Load-Store Queue Sizing for Efficient Dataflow Circuits. FPT 2022.

We Need a Load-Store Queue (LSQ)!

- Processor LSQs keep dependent memory accesses **in the original program order**

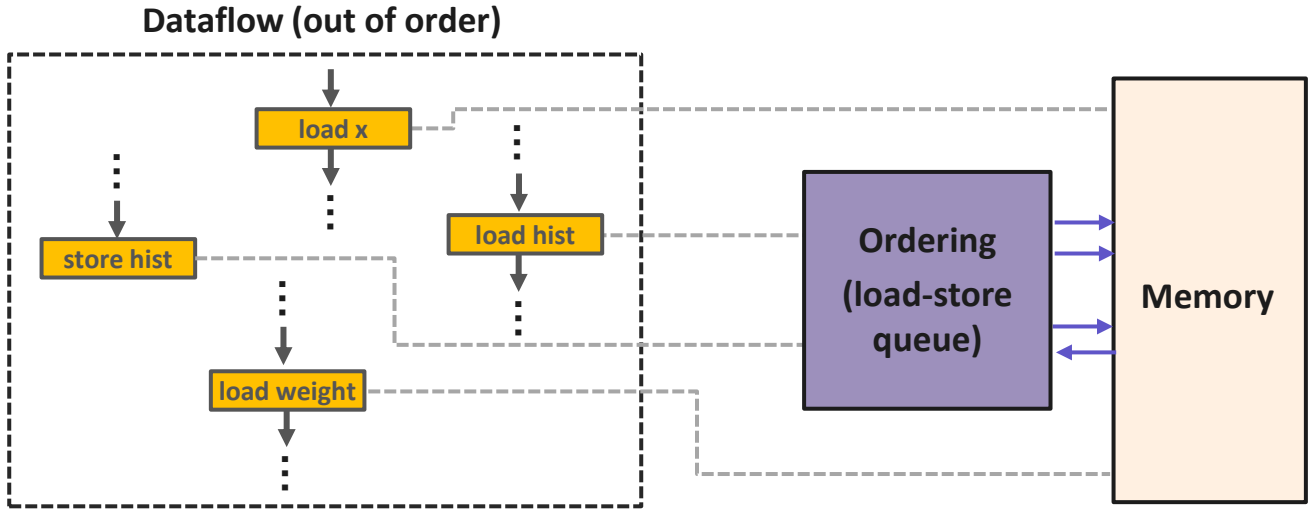
```
loop: lw $t2, 0($t4)
      lw $t3, 100($t4)
      mul $t5, $t2, $t3
      addi $t5, $t5, $t1
      sw $t5, 100($t4)
      addi $t1, $t1, 4
      bne $t6, $t1, loop
```



- **Application-specific LSQs** for dataflow circuits

LSQ placement and sizing for high throughput and low resources

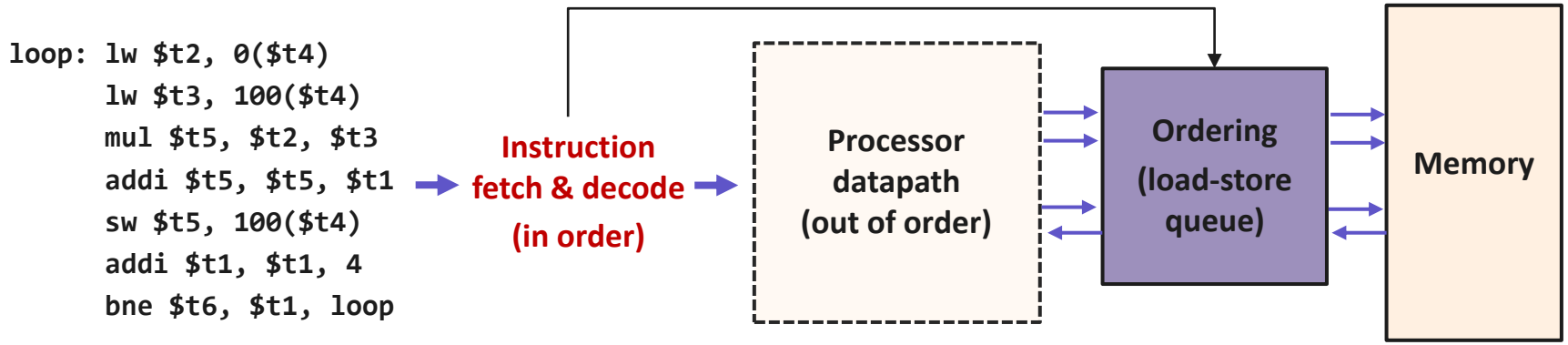
```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```



Josipović, Brisk, and Ienne. An Out-of-Order Load-Store Queue for Spatial Computing. CASES 2017 **Best Paper Award Nominee**
Liu, Rizzi, and Josipović. Load-Store Queue Sizing for Efficient Dataflow Circuits. FPT 2022.

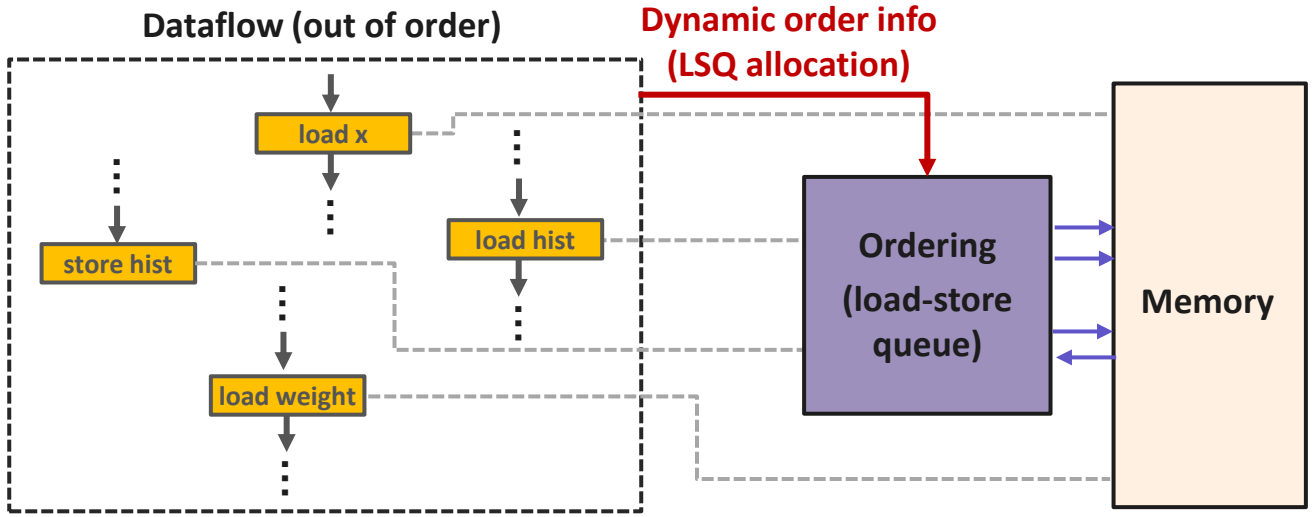
We Need a Load-Store Queue (LSQ)!

- Processor LSQs keep dependent memory accesses **in the original program order**



- **Application-specific LSQs** for dataflow circuits

Memory access ordering info devised by dataflow circuit



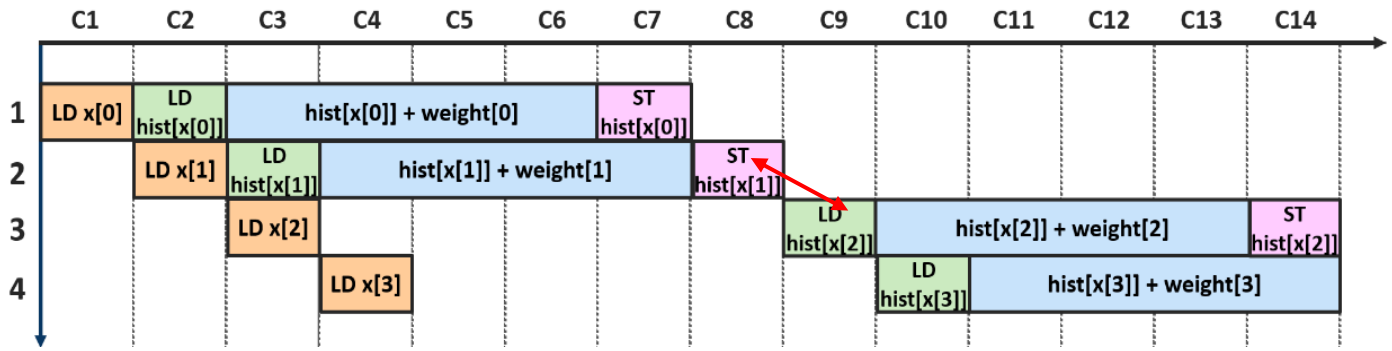
Josipović, Brisk, and Ienne. An Out-of-Order Load-Store Queue for Spatial Computing. CASES 2017 **Best Paper Award Nominee**
Liu, Rizzi, and Josipović. Load-Store Queue Sizing for Efficient Dataflow Circuits. FPT 2022.

Dataflow Circuit with the LSQ

```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

```
1: x[0]=5 → ld hist[5]; st hist[5];
2: x[1]=4 → ld hist[4]; st hist[4];
3: x[2]=4 → ld hist[4]; st hist[4];
```

RAW dependency

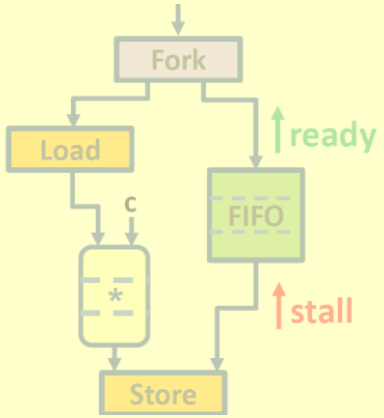


High-throughput pipeline with memory dependencies honored

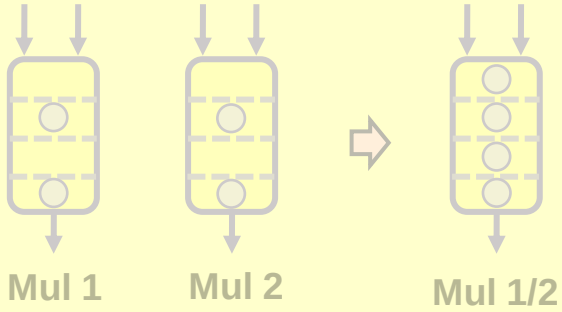
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

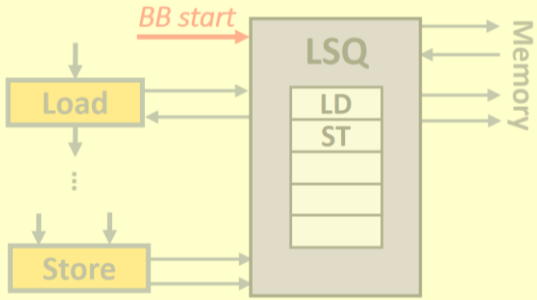


Resource sharing

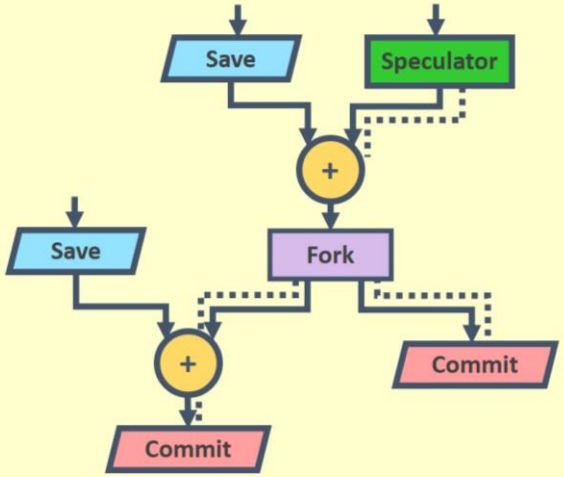


Reaping the benefits of dynamic scheduling

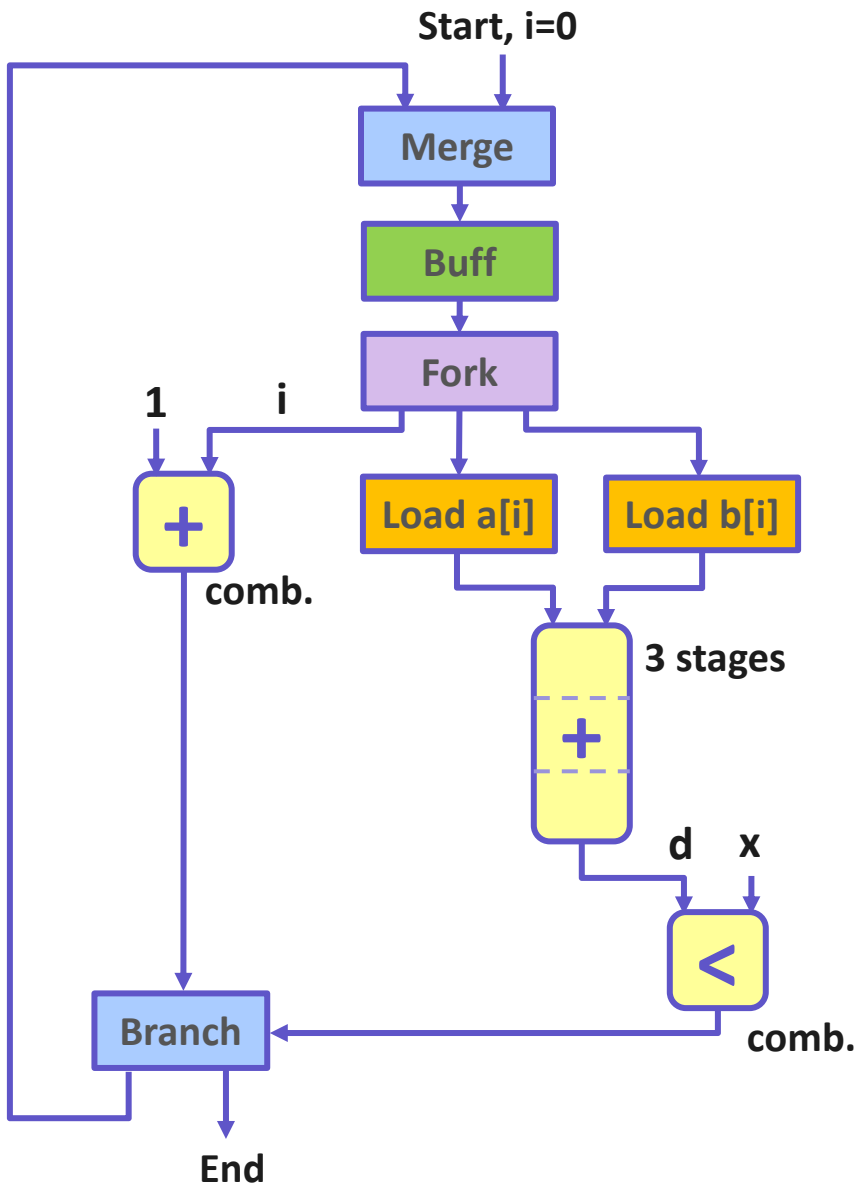
Out-of-order memory



Speculative execution

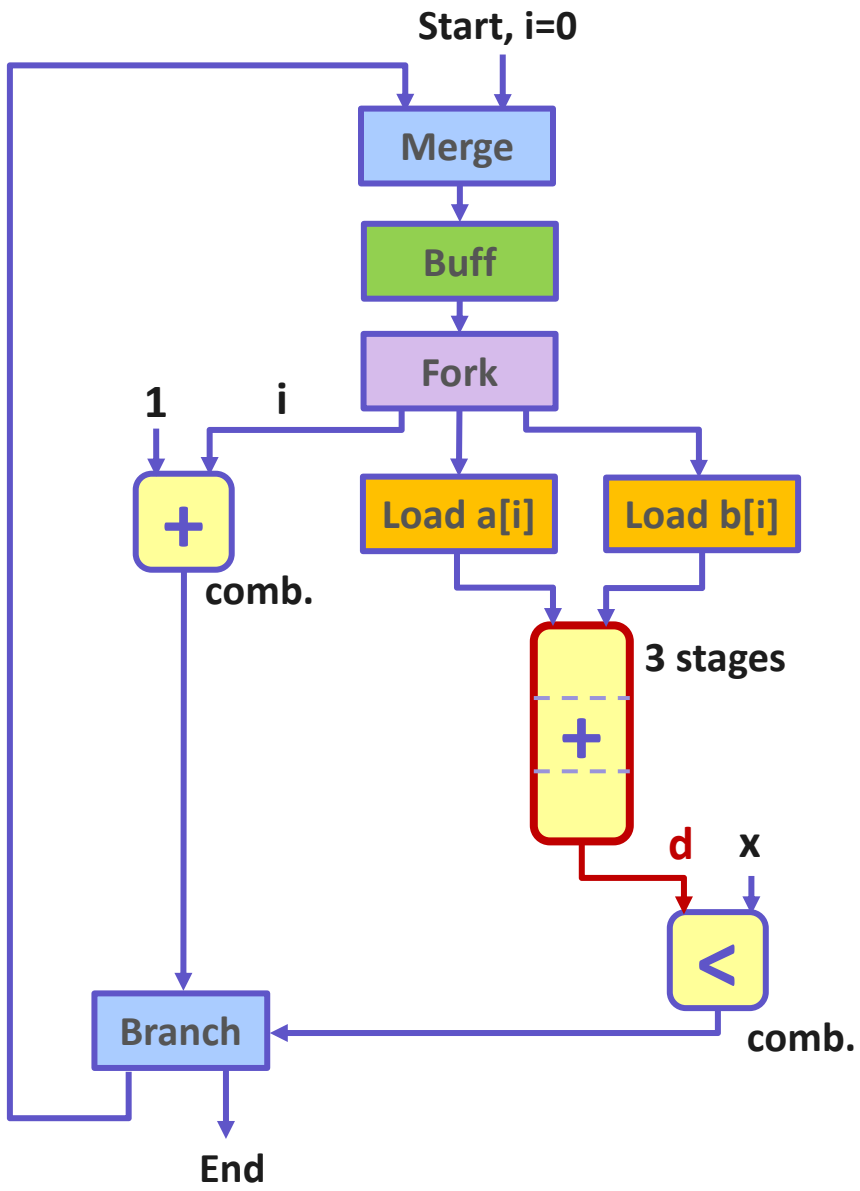


Nonspeculative Dataflow Circuit



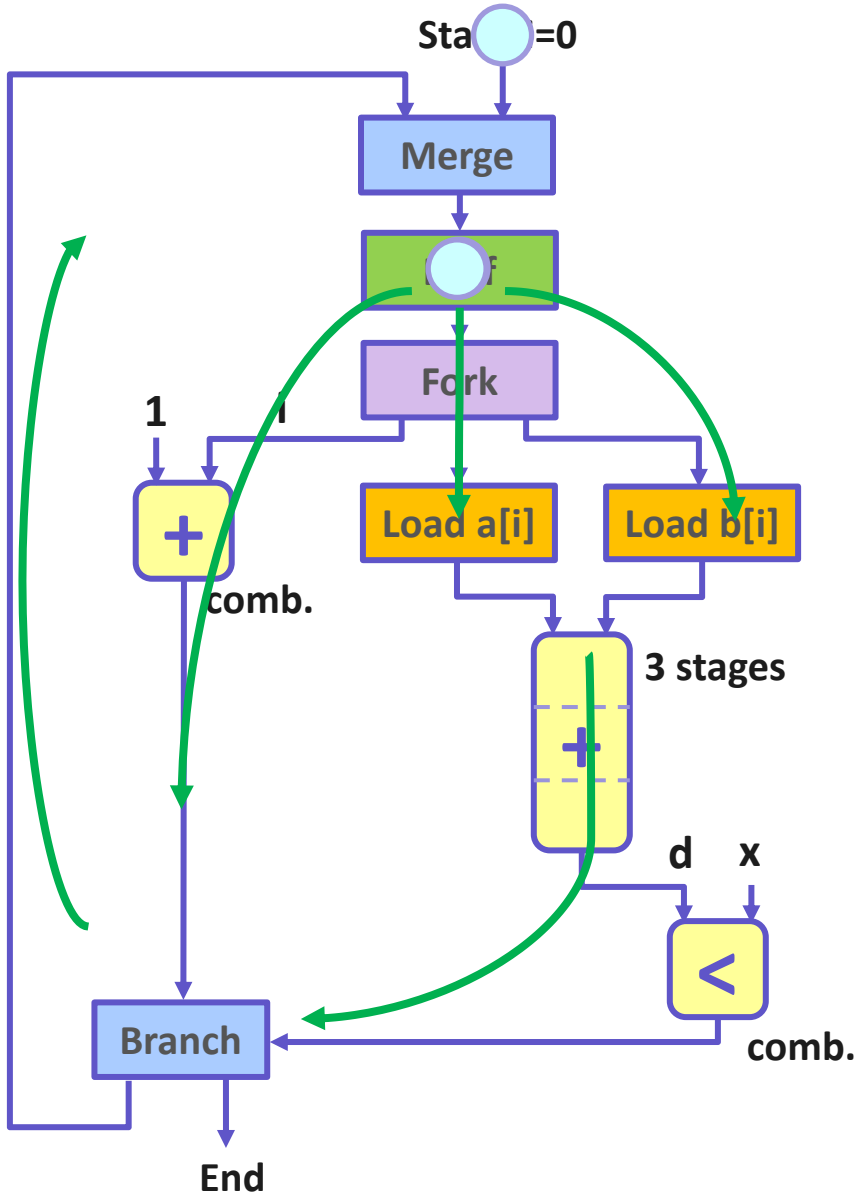

```
float d=0.0; x=100.0; int i=0;

do {
   d = a[i] + b[i];
   i++;
}
while (d<x);
```

Nonspeculative Dataflow Circuit




```
float d=0.0; x=100.0; int i=0;

do {
   d = a[i] + b[i];
   i++;
}
while (d<x);
```

Nonspeculative Dataflow Circuit



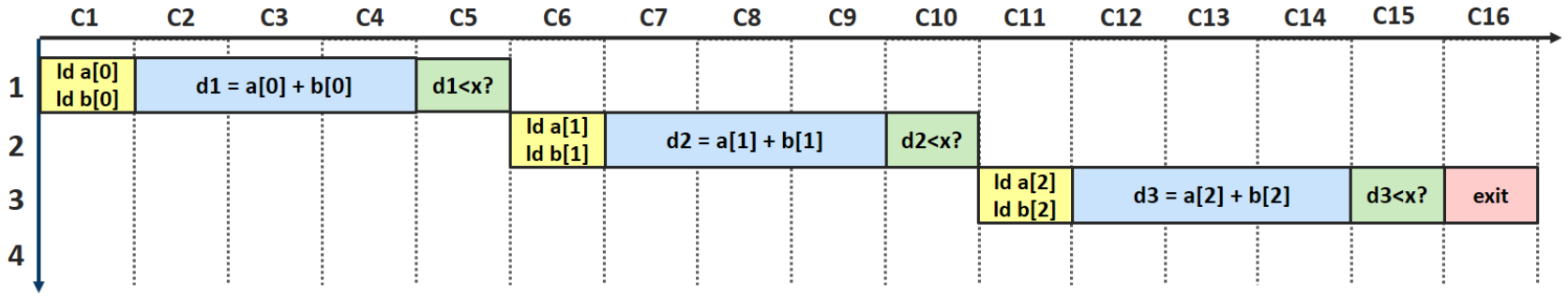

```
float d=0.0; x=100.0; int i=0;

do {
   d = a[i] + b[i];
   i++;
}
while (d<x);
```

Nonspeculative vs. Speculative System

```
float d=0.0; x=100.0; int i=0;

do {
   d = a[i] + b[i];
   i++;
}
while (d<x);
```

```
1: a[0]=50.0; b[0]=30.0
2: a[1]=40.0; b[1]=40.0
3: a[2]=50.0; b[2]=60.0 → exit
```

Nonspeculative schedule

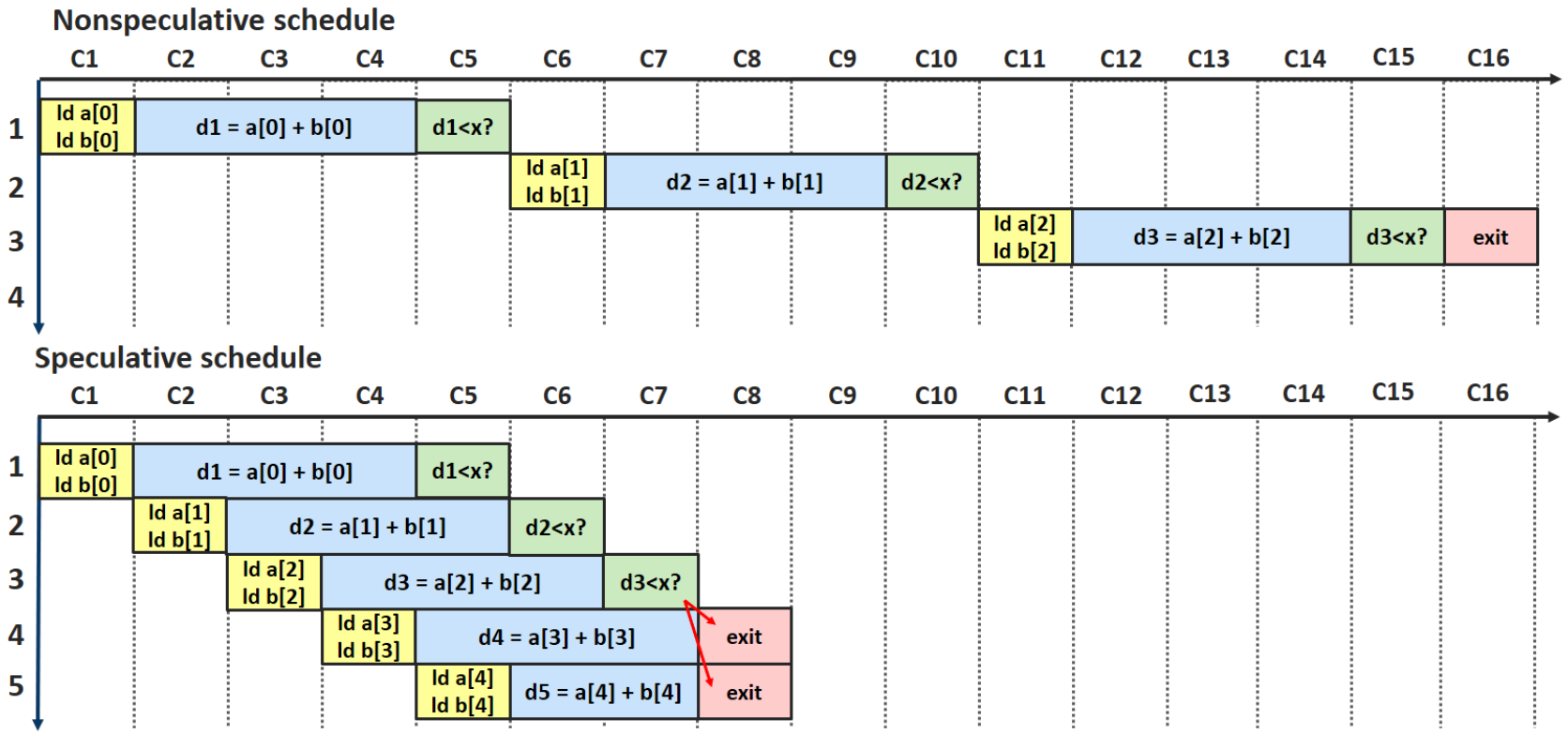


Long control flow decision prevents pipelining

Nonspeculative vs. Speculative System

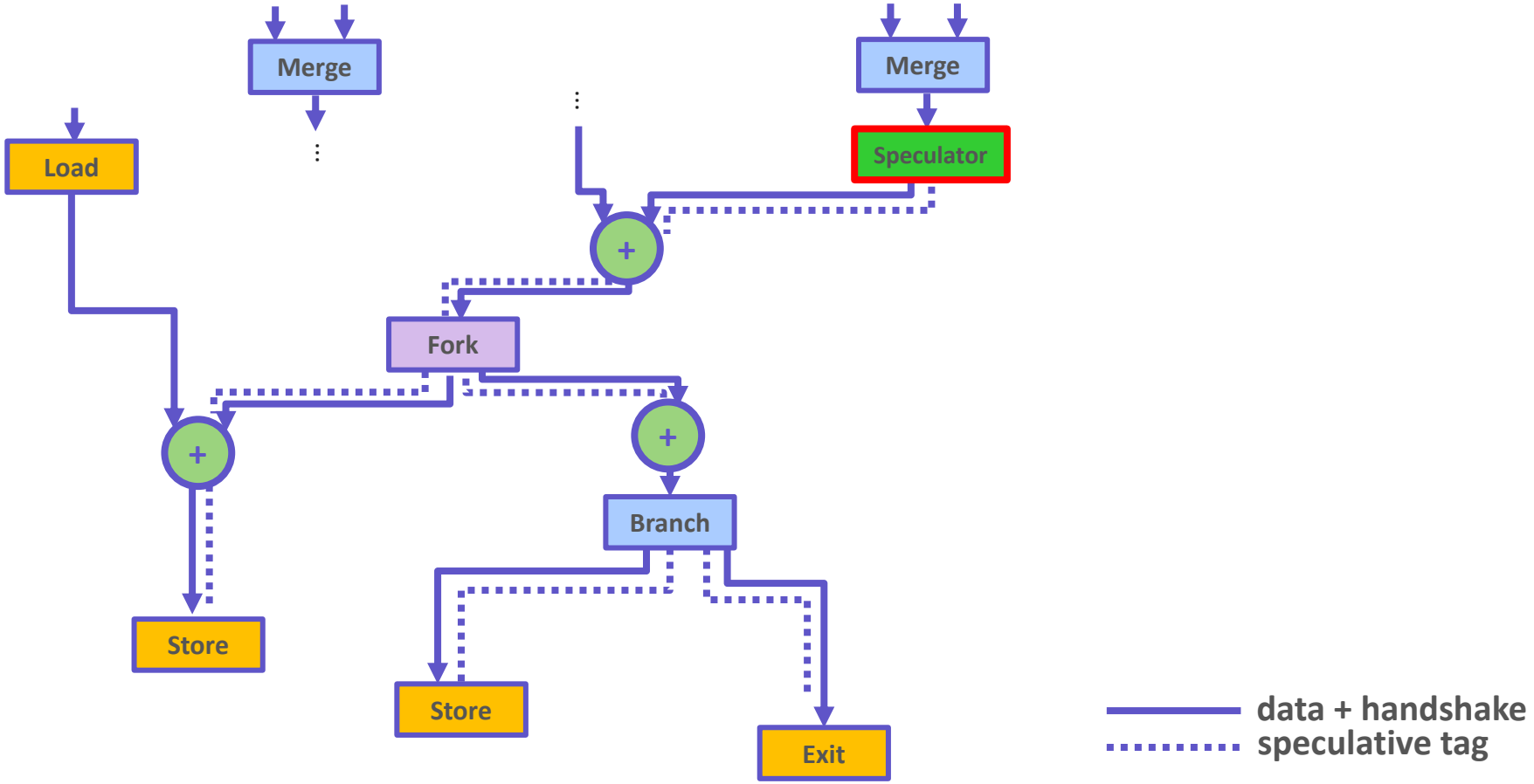
```
float d=0.0; x=100.0; int i=0;

do {
   d = a[i] + b[i];
   i++;
}
while (d<x);
```

```
1: a[0]=50.0; b[0]=30.0
2: a[1]=40.0; b[1]=40.0
3: a[2]=50.0; b[2]=60.0 → exit
```



Speculation in Dataflow Circuits

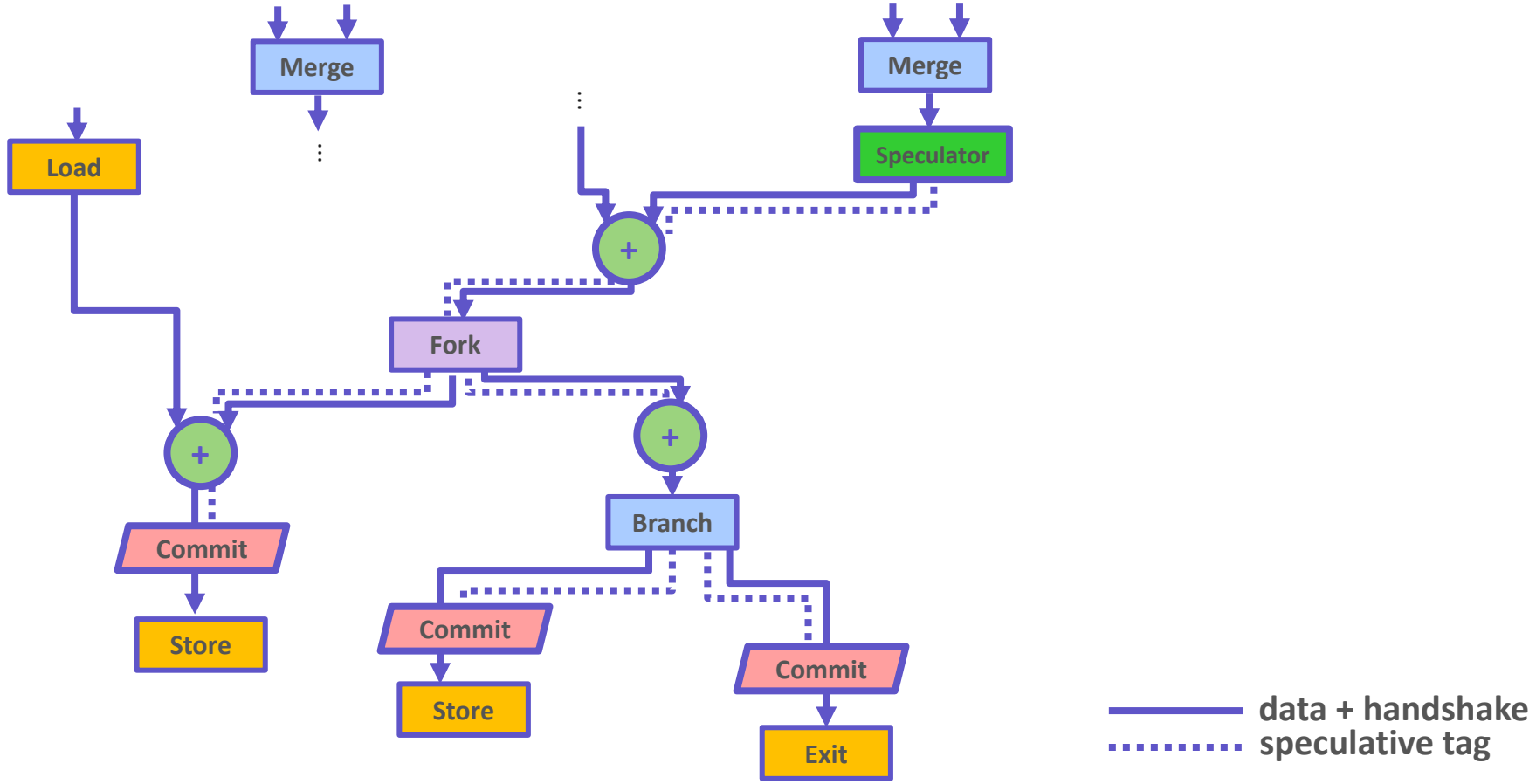
- Contain speculation in a region of the circuit delimited by special components
  - Issue speculative tokens (pieces of data which might or might not be correct)
  - Squash and replay in case of misspeculation



Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019

Speculation in Dataflow Circuits

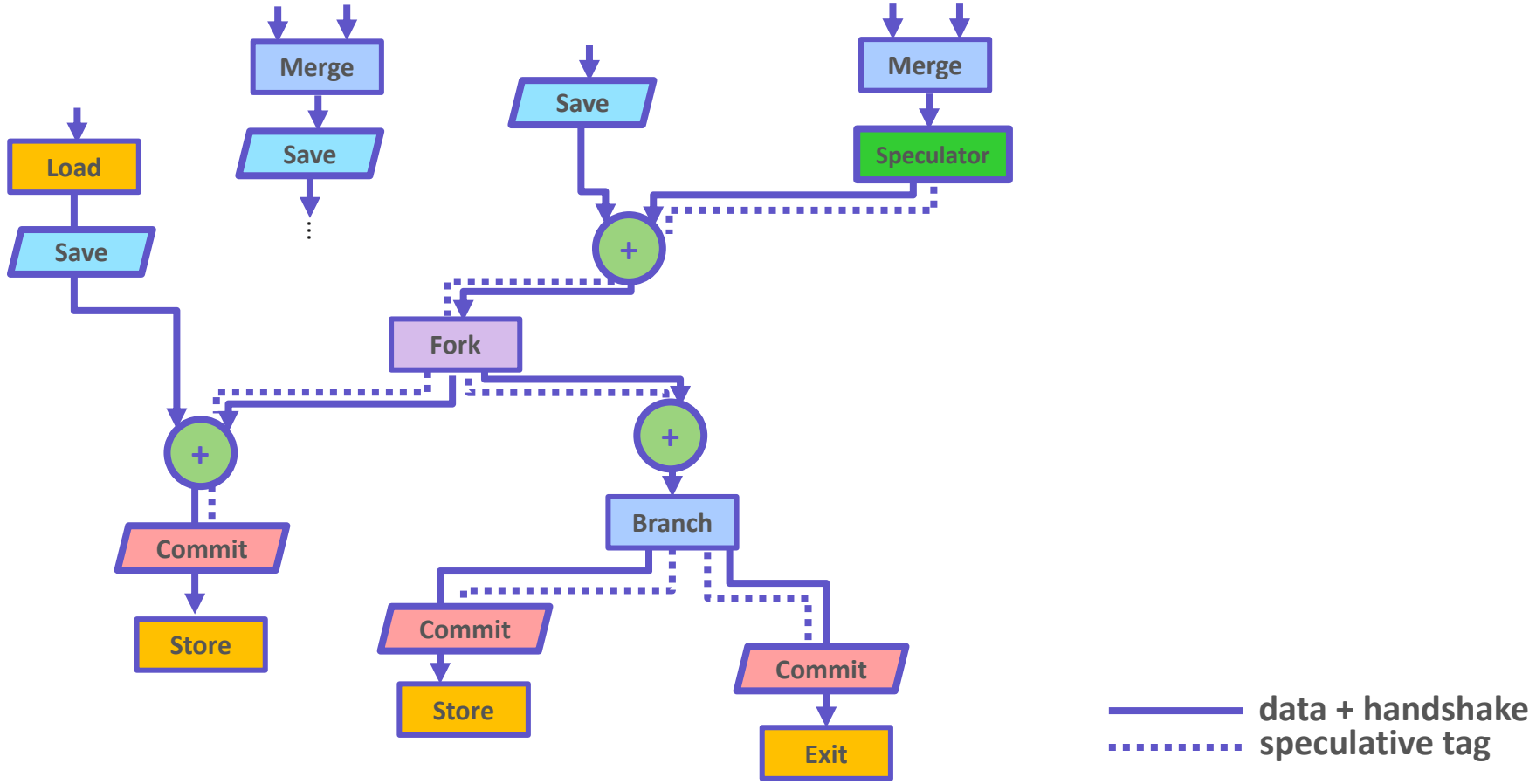
- Contain speculation in a region of the circuit delimited by special components
  - Issue speculative tokens (pieces of data which might or might not be correct)
  - Squash and replay in case of misspeculation



Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019

Speculation in Dataflow Circuits

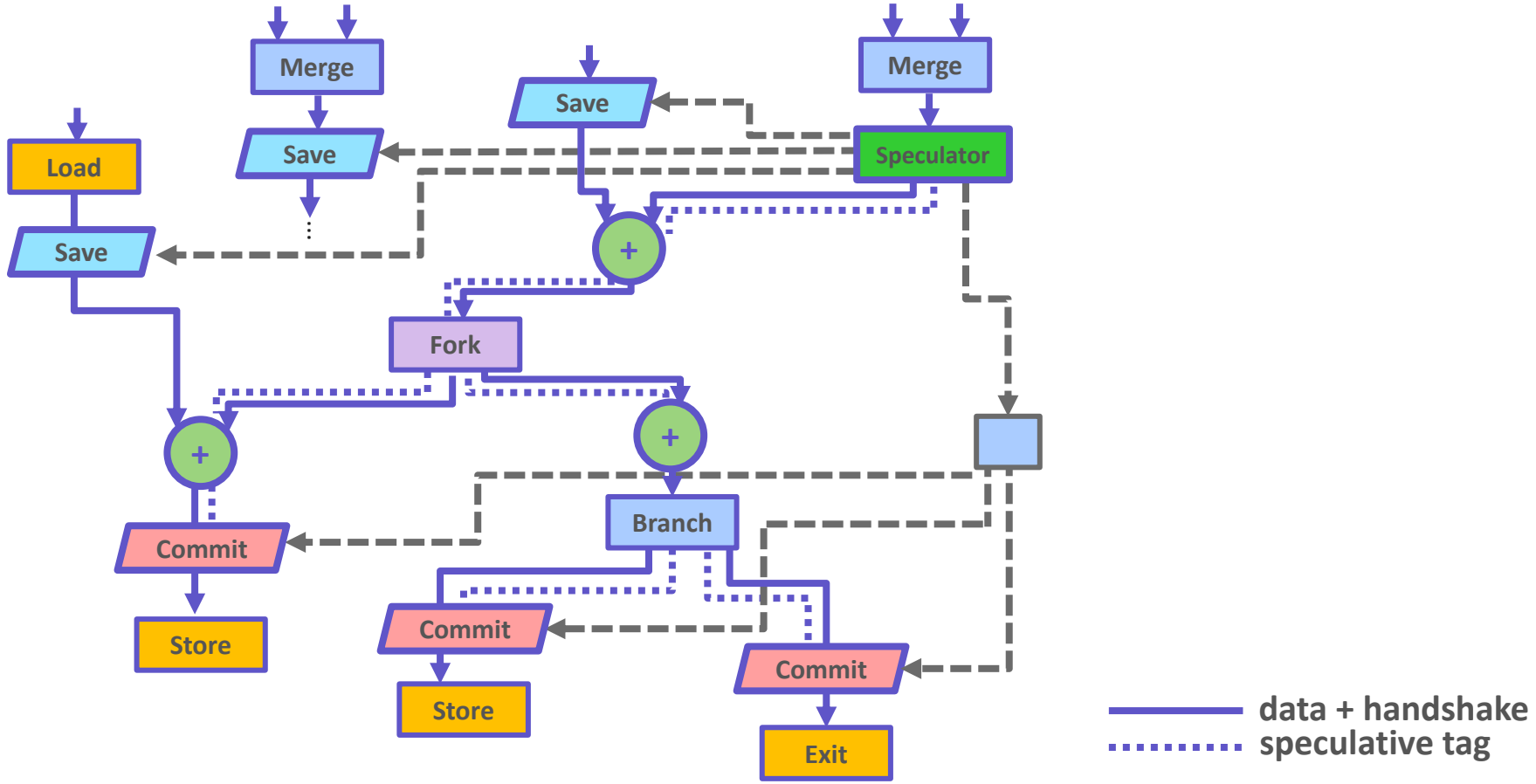
- Contain speculation in a region of the circuit delimited by special components
  - Issue speculative tokens (pieces of data which might or might not be correct)
  - Squash and replay in case of misspeculation



Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019

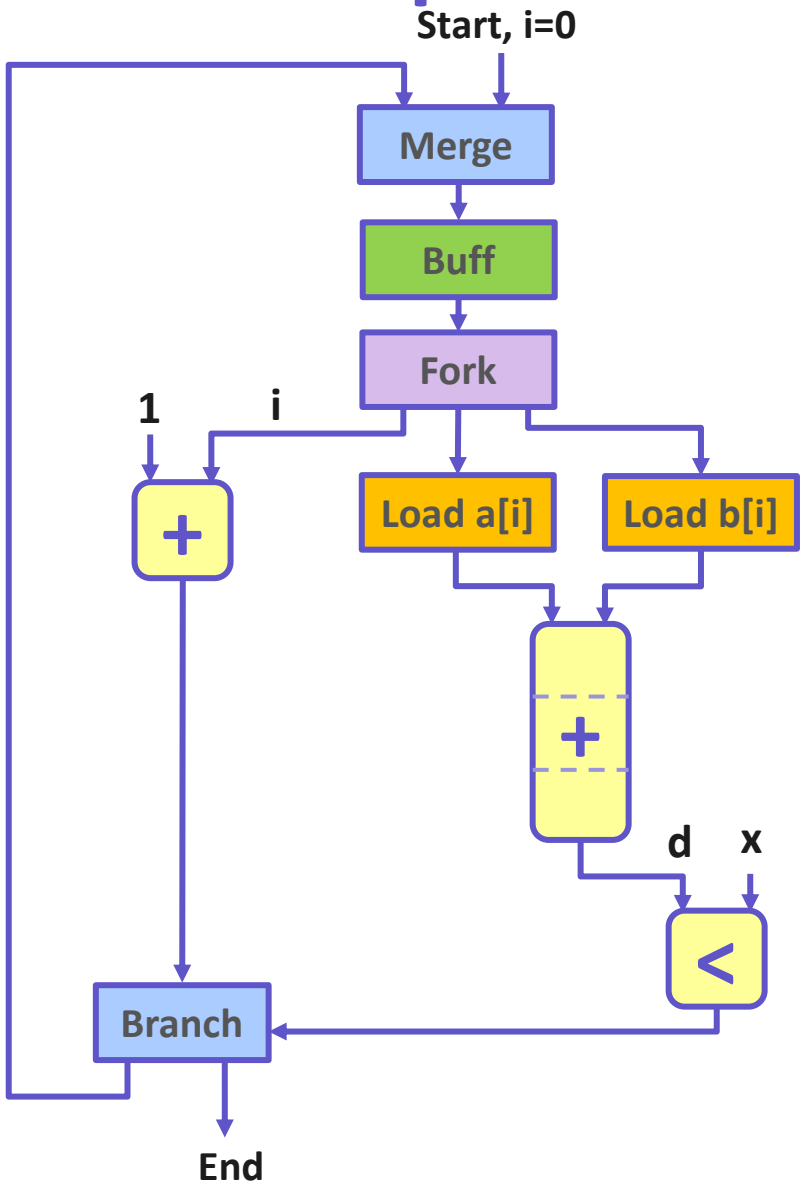
Speculation in Dataflow Circuits

- Contain speculation in a region of the circuit delimited by special components
  - Issue speculative tokens (pieces of data which might or might not be correct)
  - Squash and replay in case of misspeculation



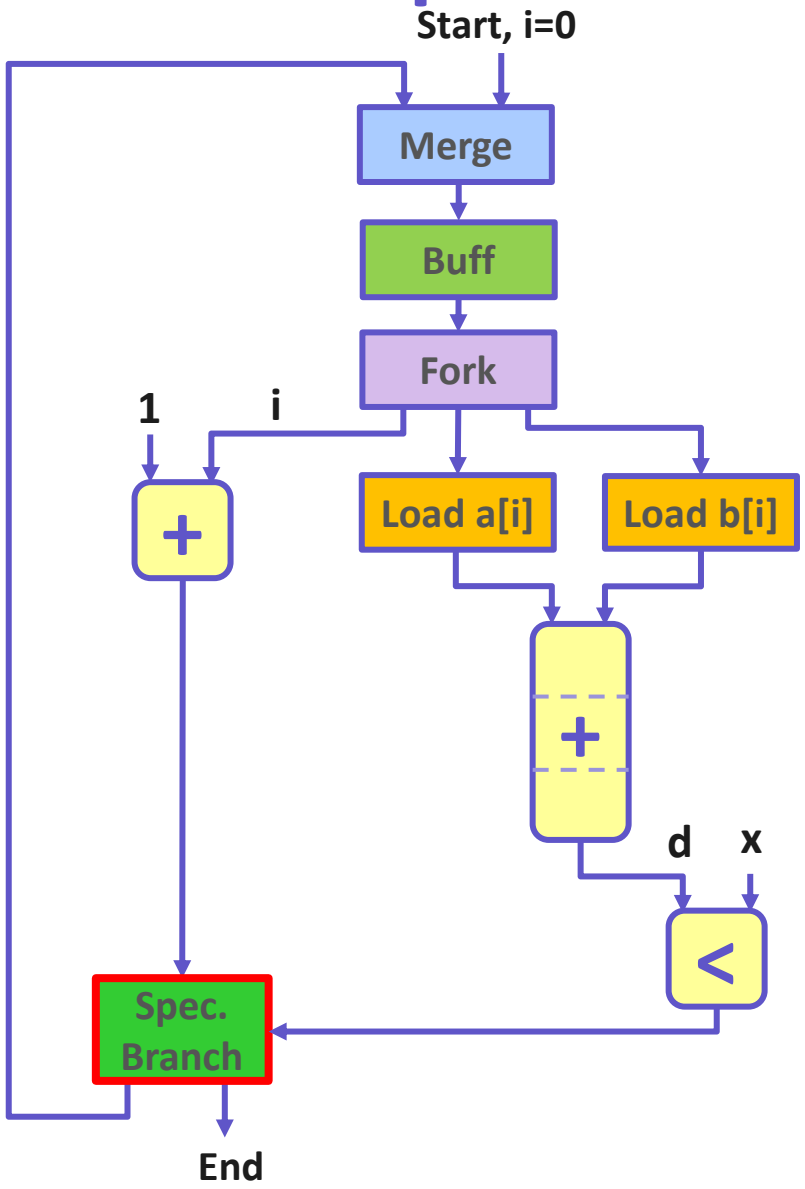
Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019

Speculative Dataflow Circuit



Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019

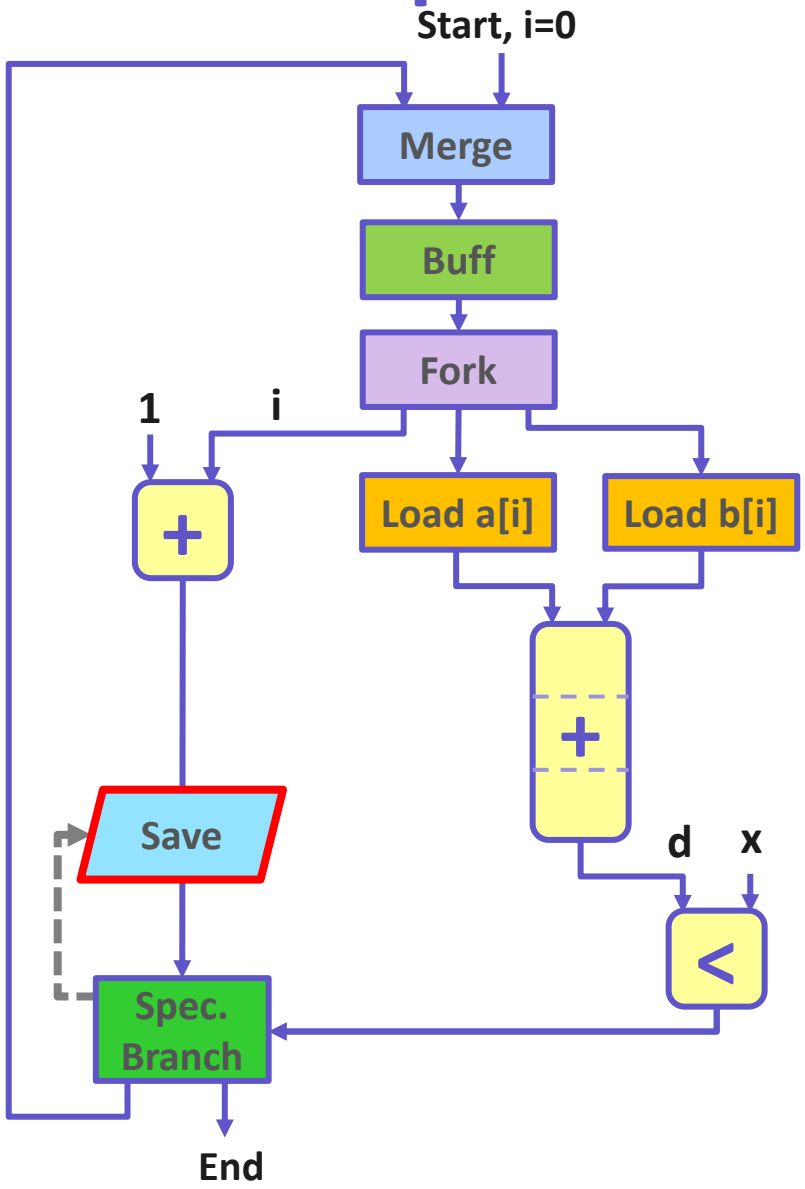
Speculative Dataflow Circuit



Speculator instead of regular Branch

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019

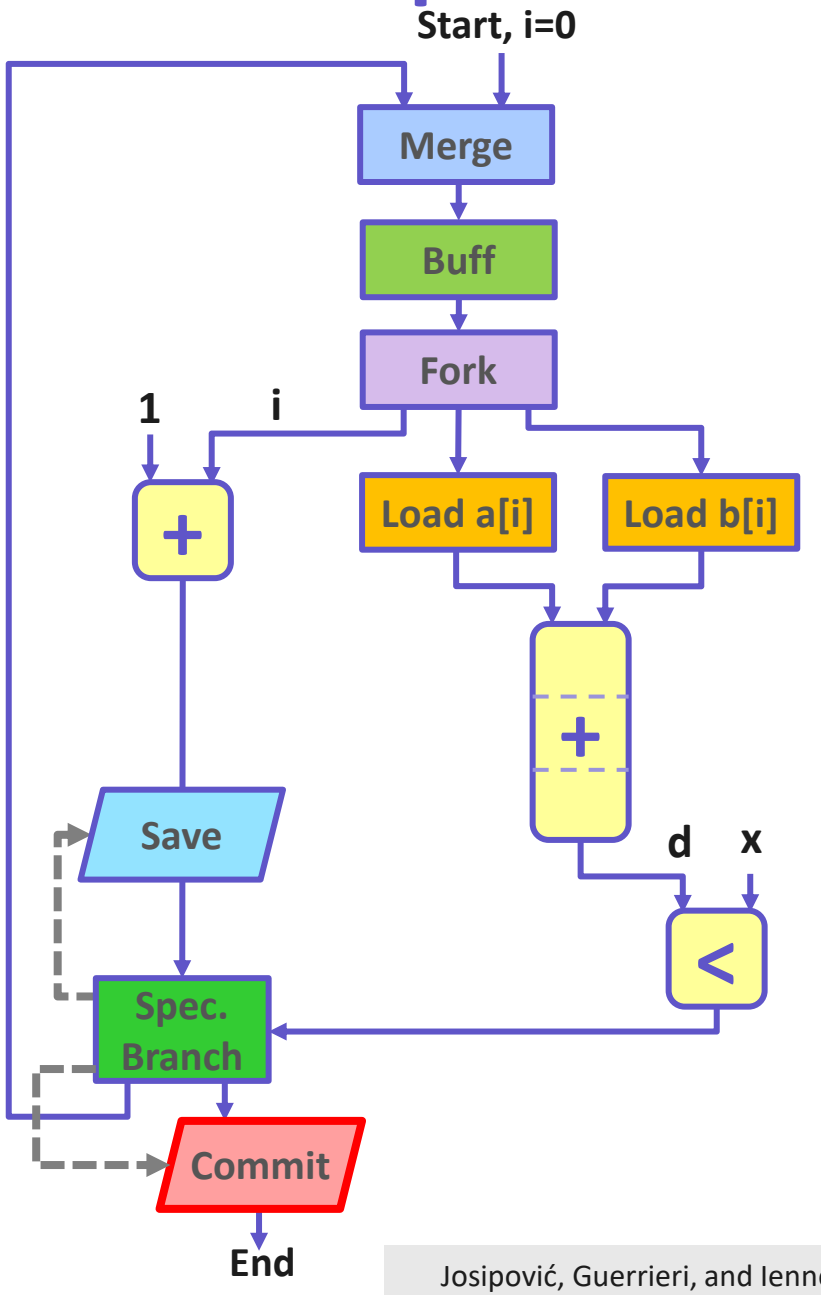
Speculative Dataflow Circuit



Input boundary:
Save units

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019

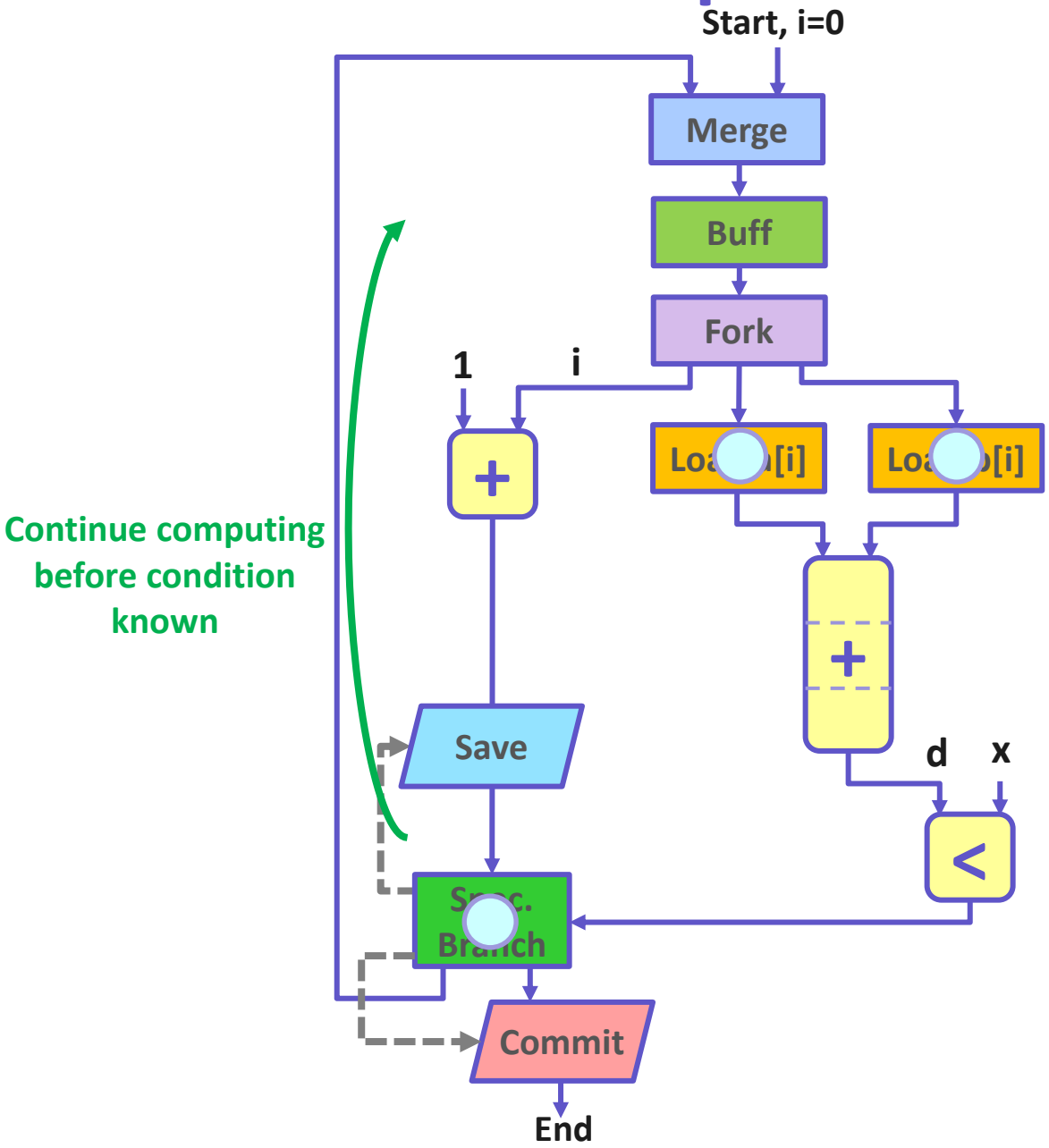
Speculative Dataflow Circuit

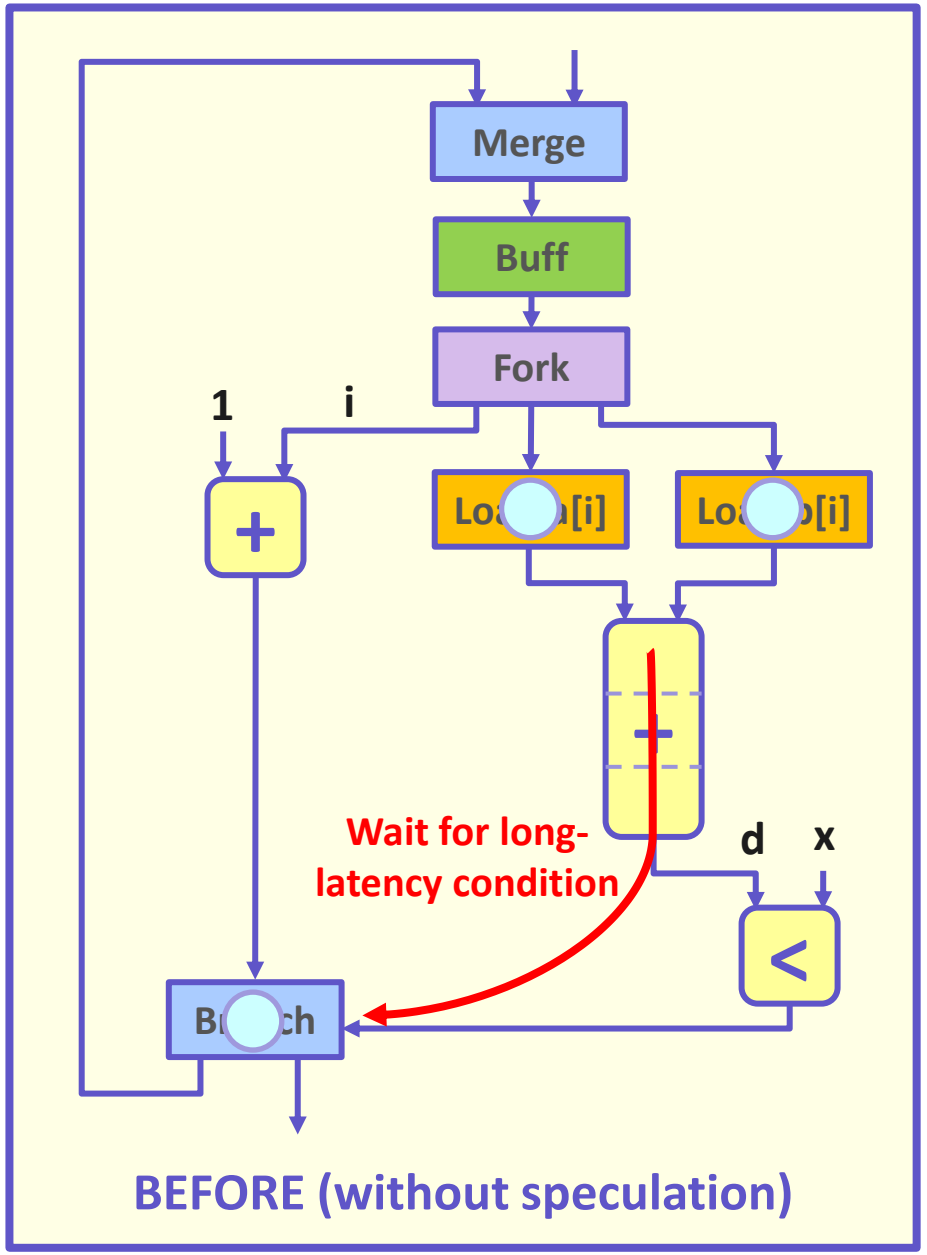


Output boundary: Commit units

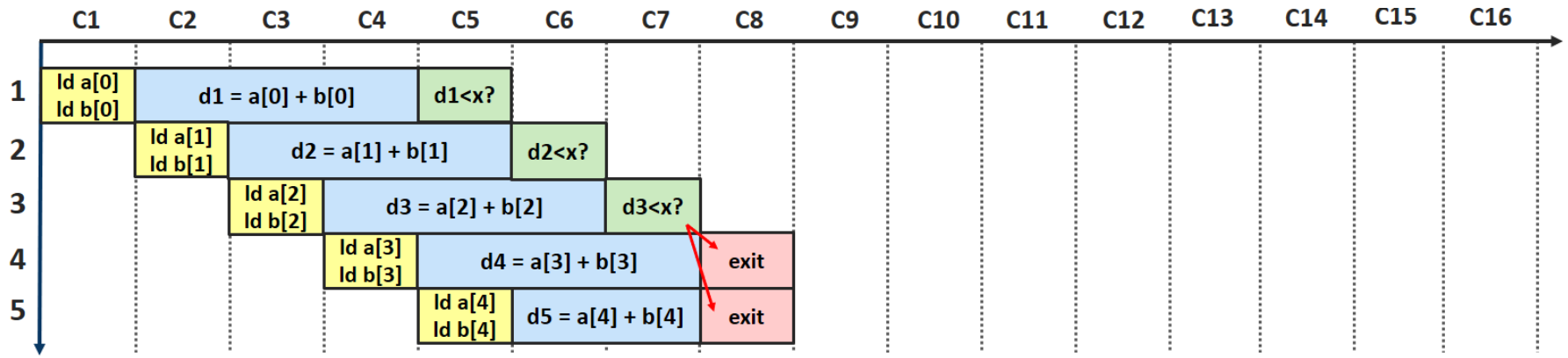
Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019

Speculative Dataflow Circuit





Speculative Dataflow Circuit



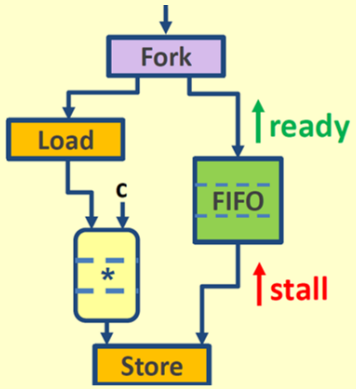
High-throughput speculative pipeline

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019

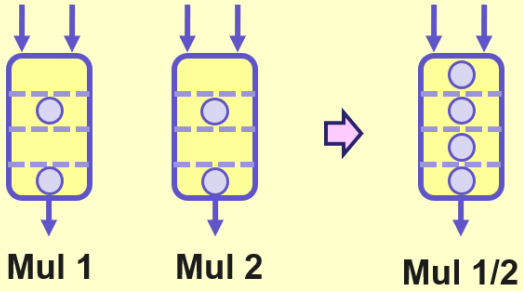
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

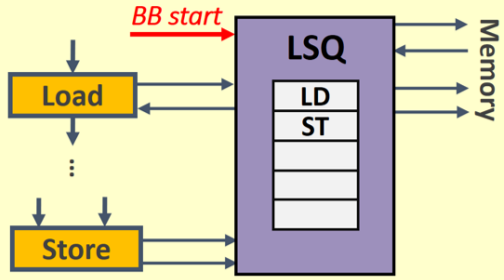


Resource sharing

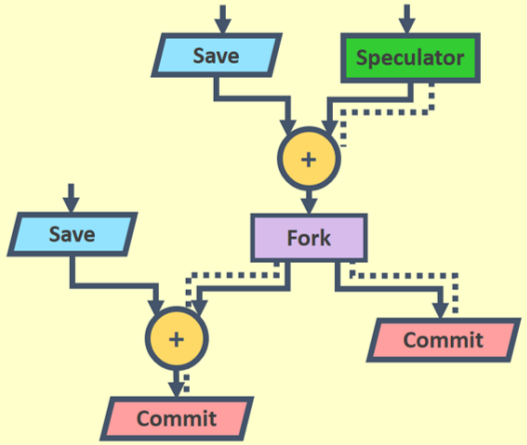


Reaping the benefits of dynamic scheduling

Out-of-order memory



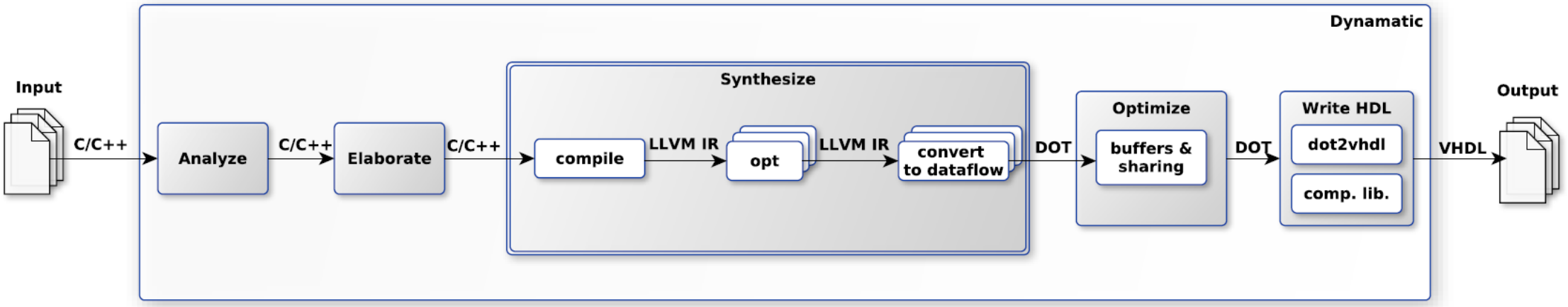
Speculative execution



Static HLS vs. dynamic HLS?

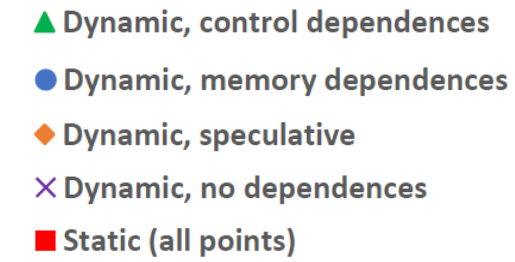
Static vs. Dynamic HLS

- **Dynamatic:** an open-source HLS compiler



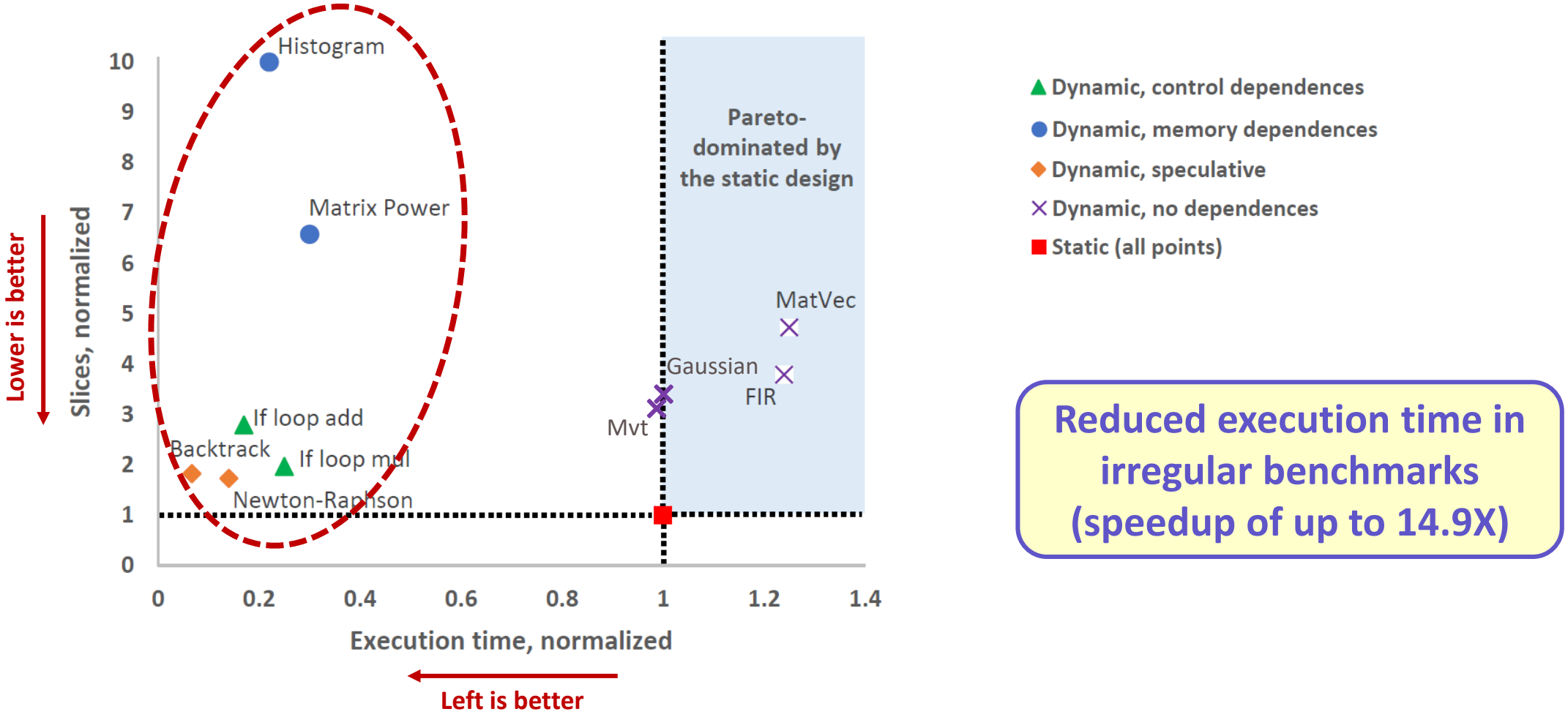
Static vs. Dynamic HLS

- **Dynamatic:** an open-source HLS compiler
- Resource utilization and execution time of the dataflow designs, **normalized to the corresponding static designs** produced by Vivado HLS



Static vs. Dynamic HLS

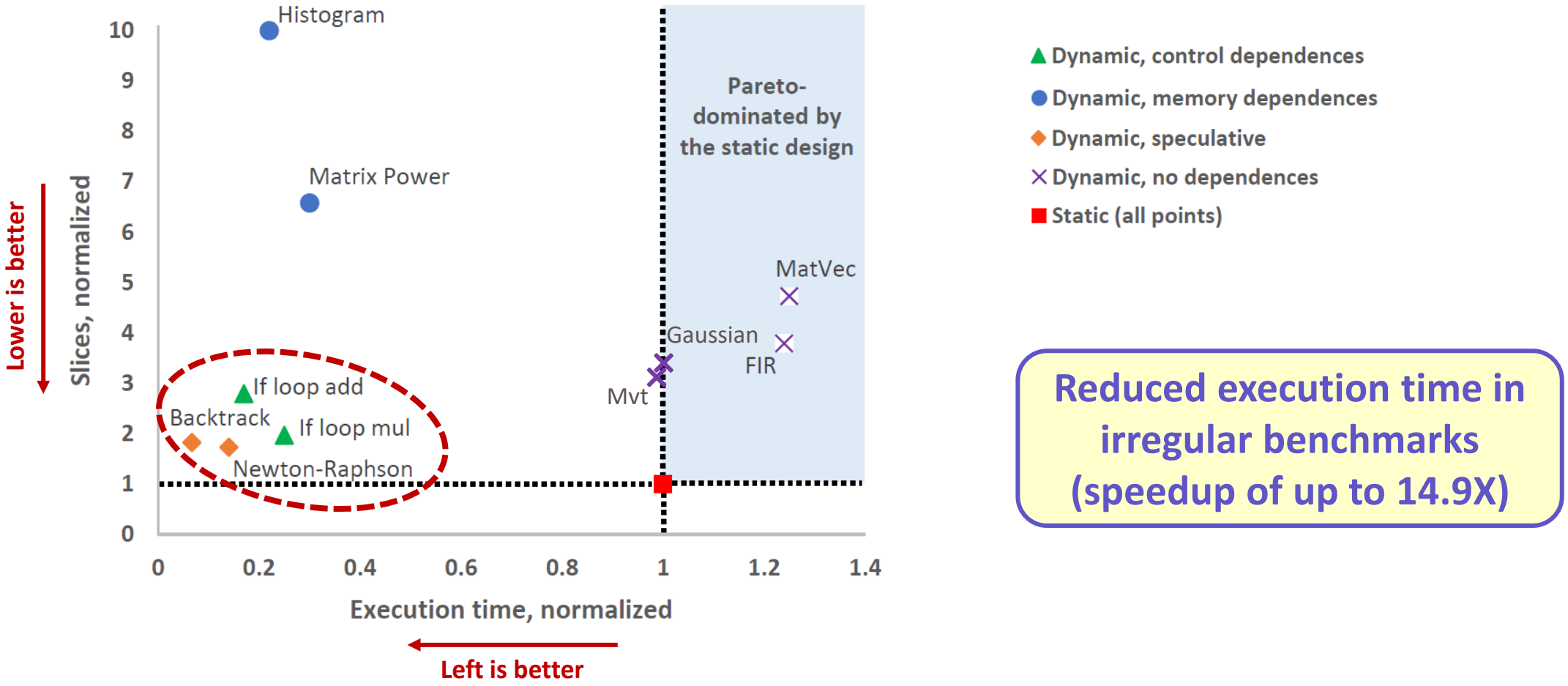
- **Dynamatic:** an open-source HLS compiler
- Resource utilization and execution time of the dataflow designs, **normalized to the corresponding static designs** produced by Vivado HLS



Reduced execution time in irregular benchmarks (speedup of up to 14.9X)

Static vs. Dynamic HLS

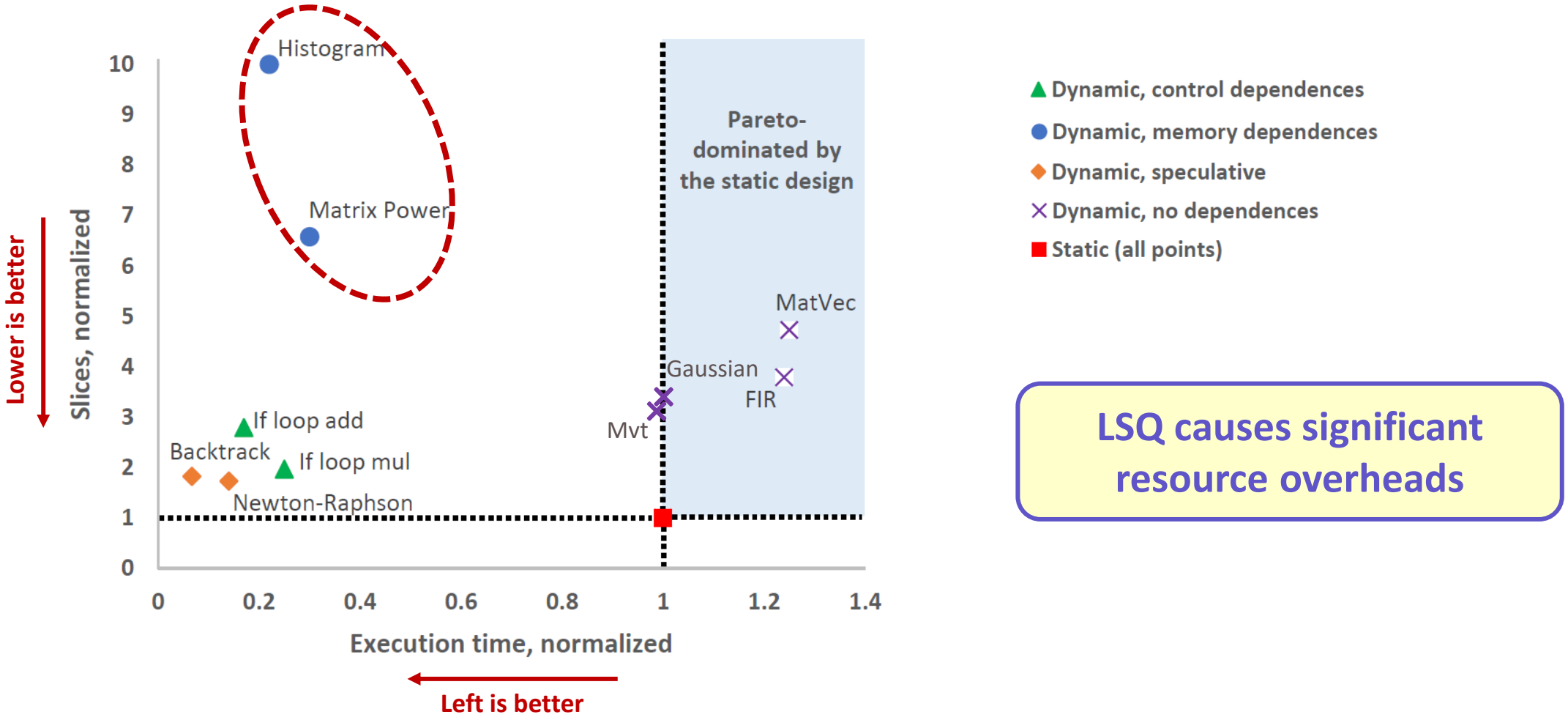
- **Dynamatic:** an open-source HLS compiler
- Resource utilization and execution time of the dataflow designs, **normalized to the corresponding static designs** produced by Vivado HLS



Reduced execution time in irregular benchmarks (speedup of up to 14.9X)

Static vs. Dynamic HLS

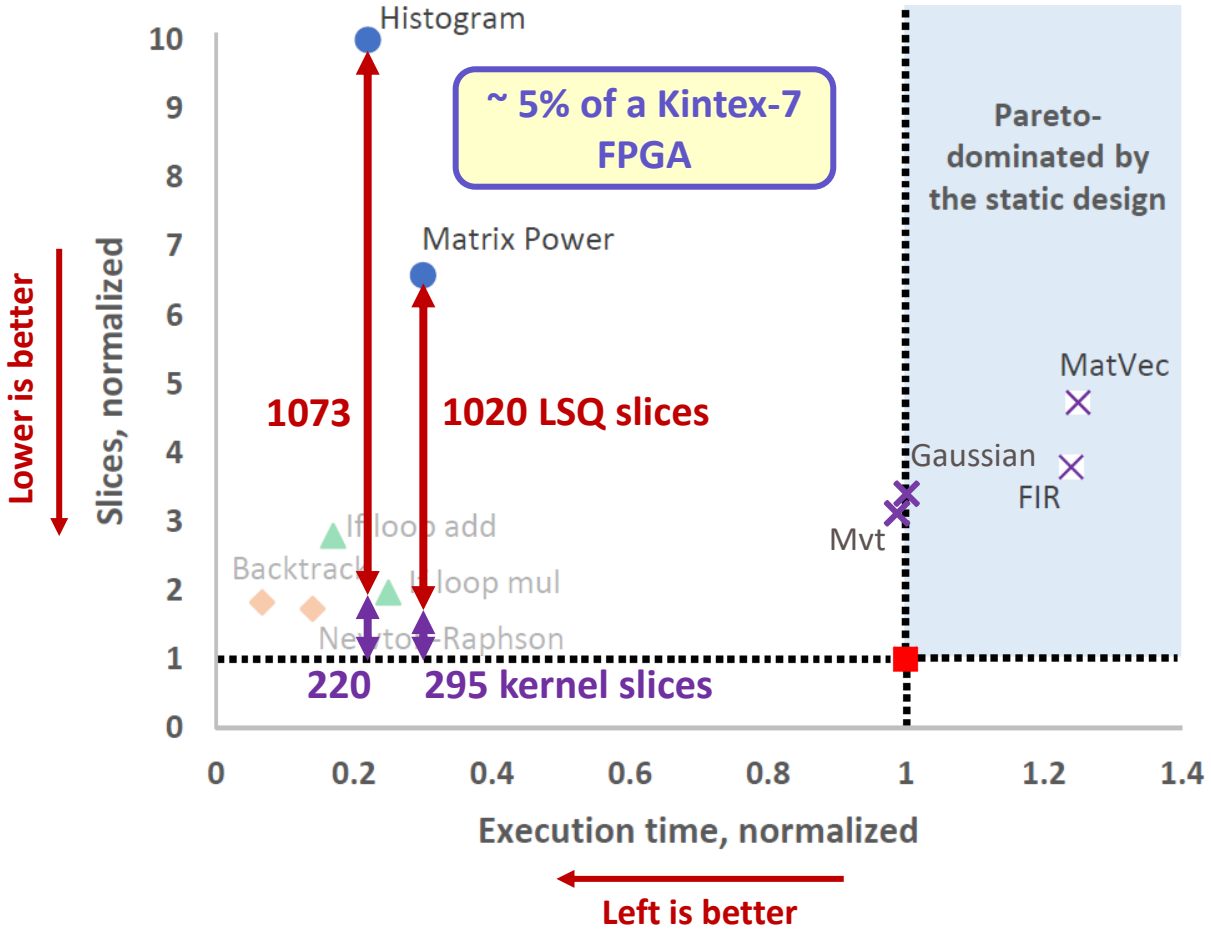
- **Dynamatic:** an open-source HLS compiler
- Resource utilization and execution time of the dataflow designs, **normalized to the corresponding static designs** produced by Vivado HLS

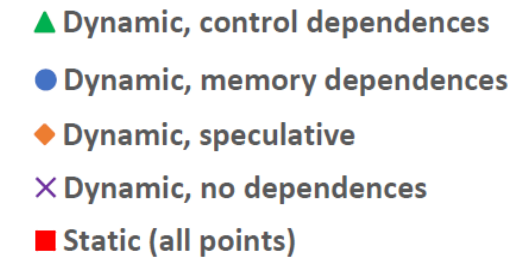


LSQ causes significant resource overheads

Static vs. Dynamic HLS

- **Dynamatic:** an open-source HLS compiler
- Resource utilization and execution time of the dataflow designs, **normalized to the corresponding static designs** produced by Vivado HLS

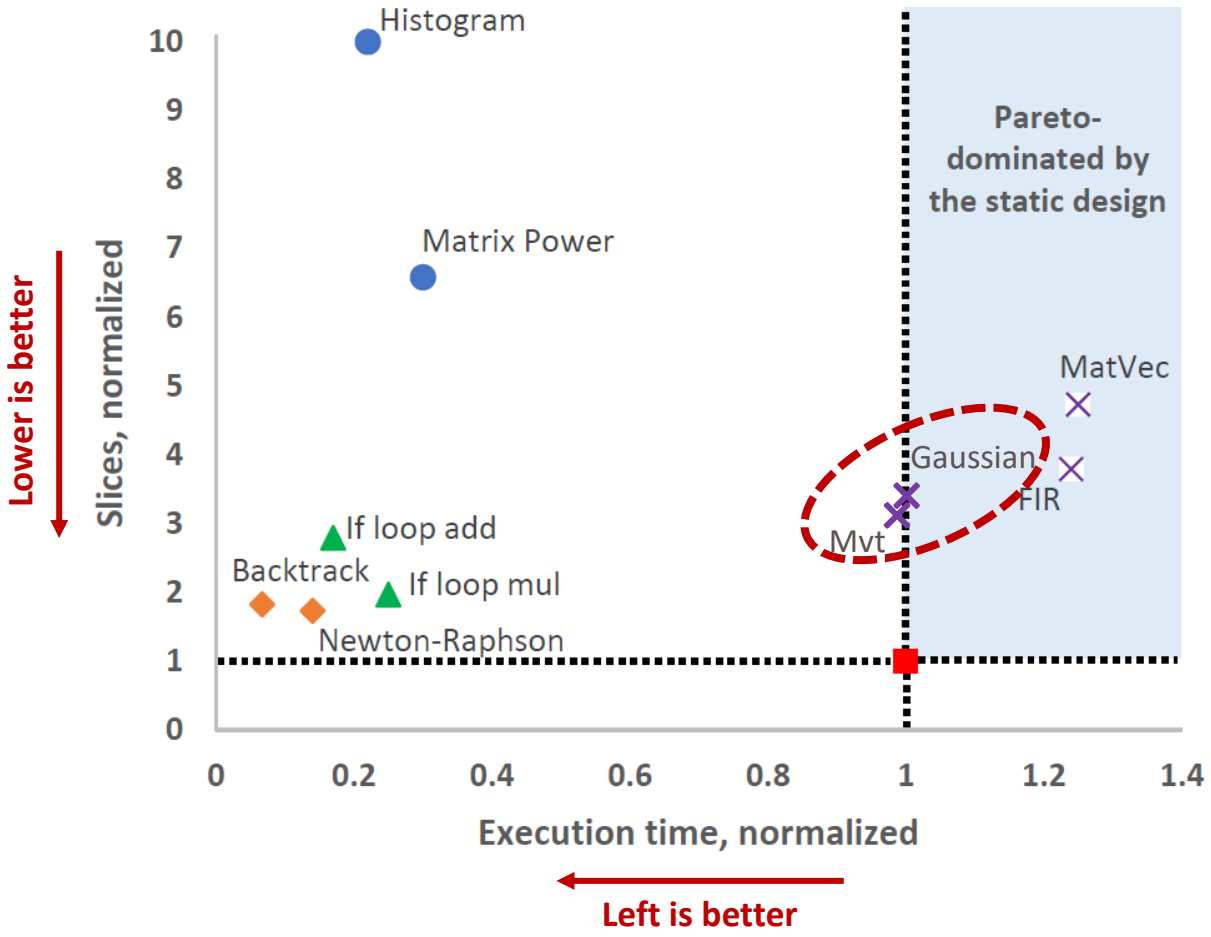


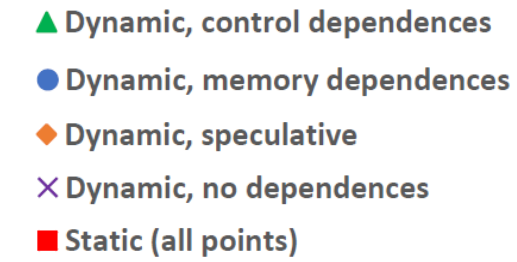


LSQ causes significant resource overheads

Static vs. Dynamic HLS

- **Dynamatic:** an open-source HLS compiler
- Resource utilization and execution time of the dataflow designs, **normalized to the corresponding static designs** produced by Vivado HLS

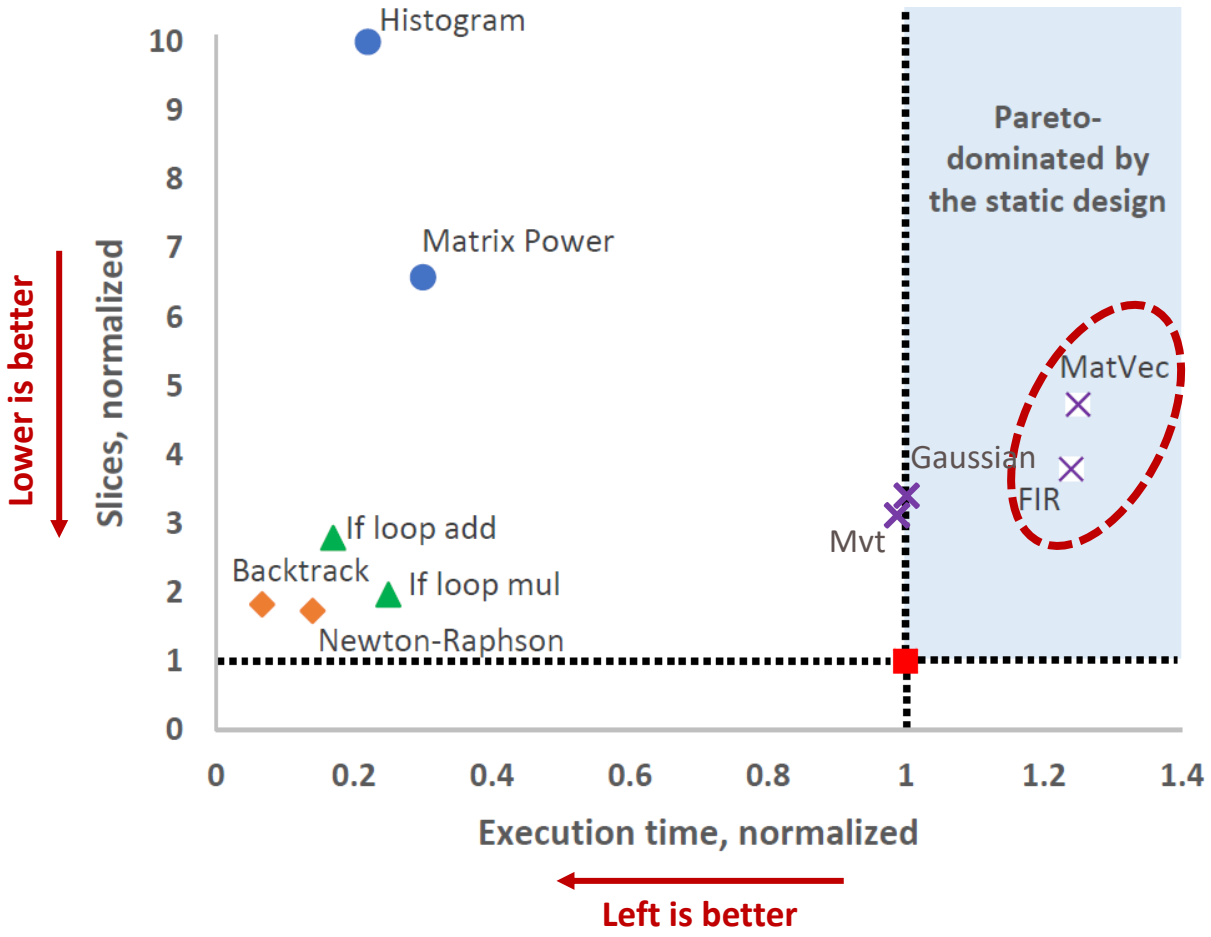


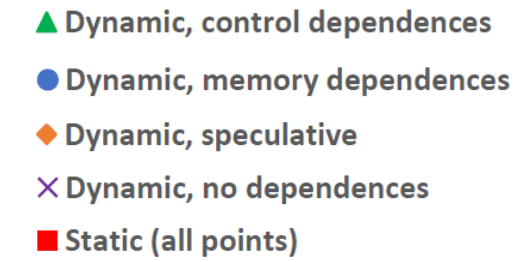


Static and dynamic HLS have the same pipelining capabilities

Static vs. Dynamic HLS

- **Dynamatic:** an open-source HLS compiler
- Resource utilization and execution time of the dataflow designs, **normalized to the corresponding static designs** produced by Vivado HLS





Regular benchmarks are Pareto-dominated due to CP increase

Static vs. Dynamic Scheduling

	Statically Scheduled → "Compiler does the job"	**Dynamically Scheduled** → "Hardware does the job"
Computer Architecture	**VLIW Processors**	**Out-of-Order Superscalar Processors**
High-Level Synthesis	**Traditional HLS**	**Dataflow circuits**

DSP-oriented applications

General-purpose code (new applications and users)

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant
for software programmers

HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

A different way to go about HLS
(generating dynamically scheduled circuits from C code)

Bridging the Gap Between Software and Hardware

SW

HW

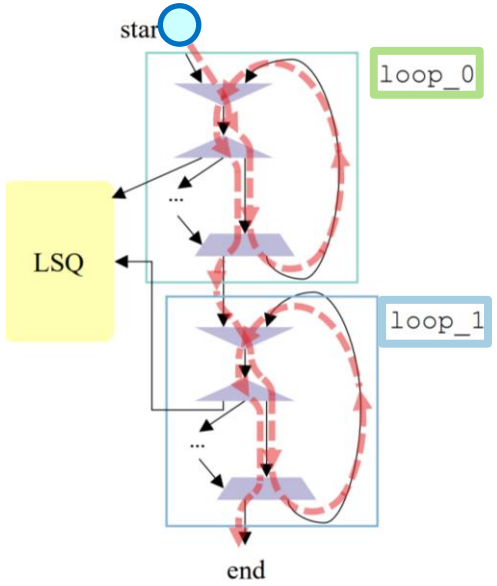
HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

Sequential C-based synthesis still **limits achievable parallelism**



Elakhras, Guerrieri, Josipović, and Ienne. Unleashing Parallelism in Elastic Circuits with Faster Token Delivery. FPL 2022 **Best Paper Award Nominee**
Cheng, Josipović, Wickerson, and Constantinides. Dynamic Inter-Block Scheduling for HLS. FPL 2022 **Best Paper Award Nominee**

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

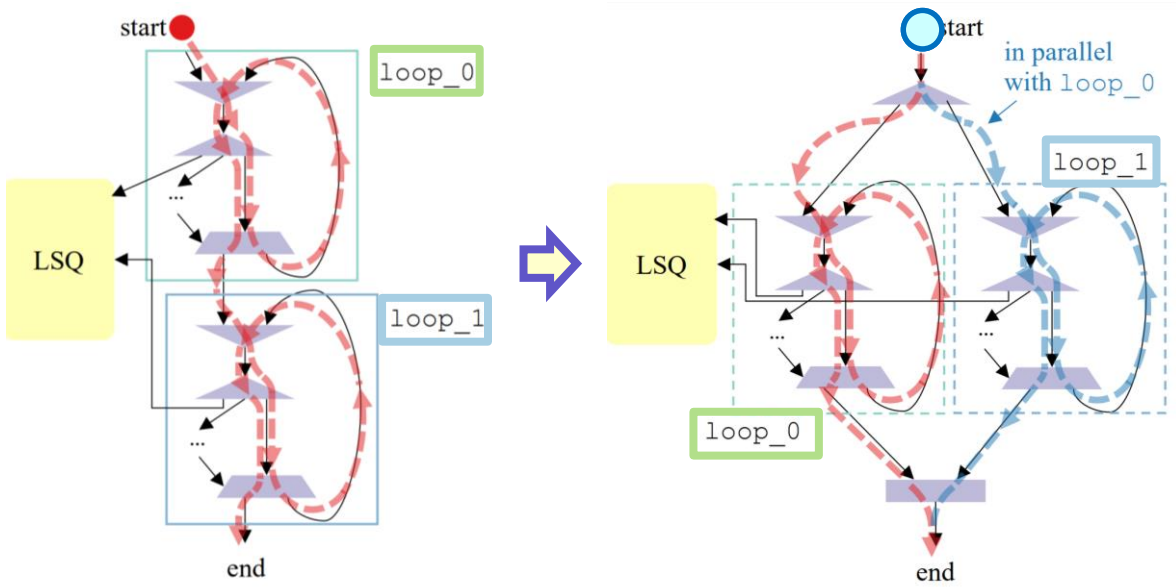
HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

Sequential C-based synthesis still **limits achievable parallelism**



New programming models and compiler techniques for **irregular parallelism**

Elakhras, Guerrieri, Josipović, and Ienne. Unleashing Parallelism in Elastic Circuits with Faster Token Delivery. FPL 2022 **Best Paper Award Nominee**
Cheng, Josipović, Wickerson, and Constantinides. Dynamic Inter-Block Scheduling for HLS. FPL 2022 **Best Paper Award Nominee**

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

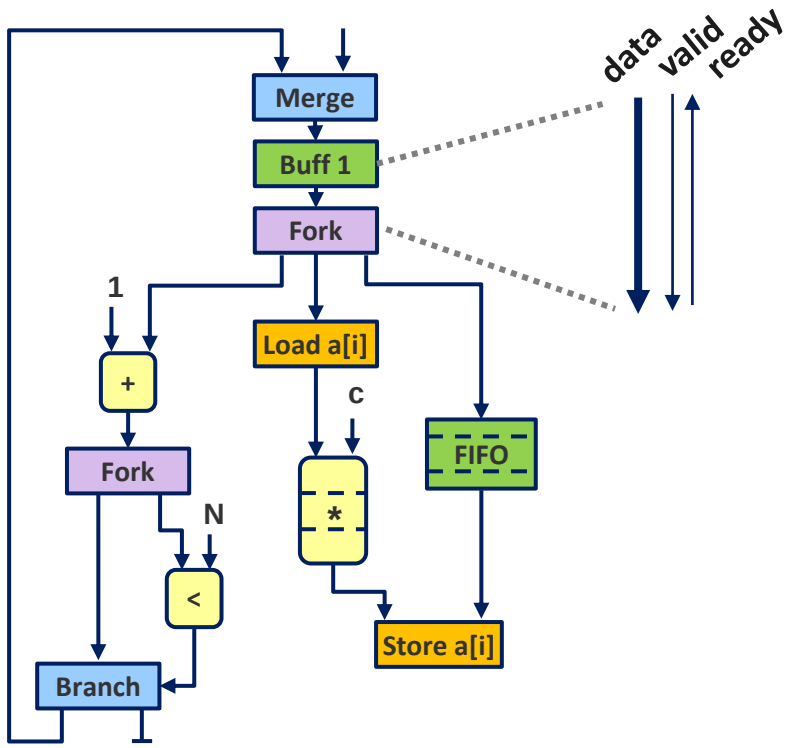
HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

Functional verification of HLS circuits using hardware simulation → inefficient and limited




```
for (i=0; i<N; i++) {
   a[i] = a[i]*c;
}
```

Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. FPGA 2023.
Xu and Josipović. Automatic inductive invariant generation for scalable dataflow circuit verification. ICCAD 2023.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

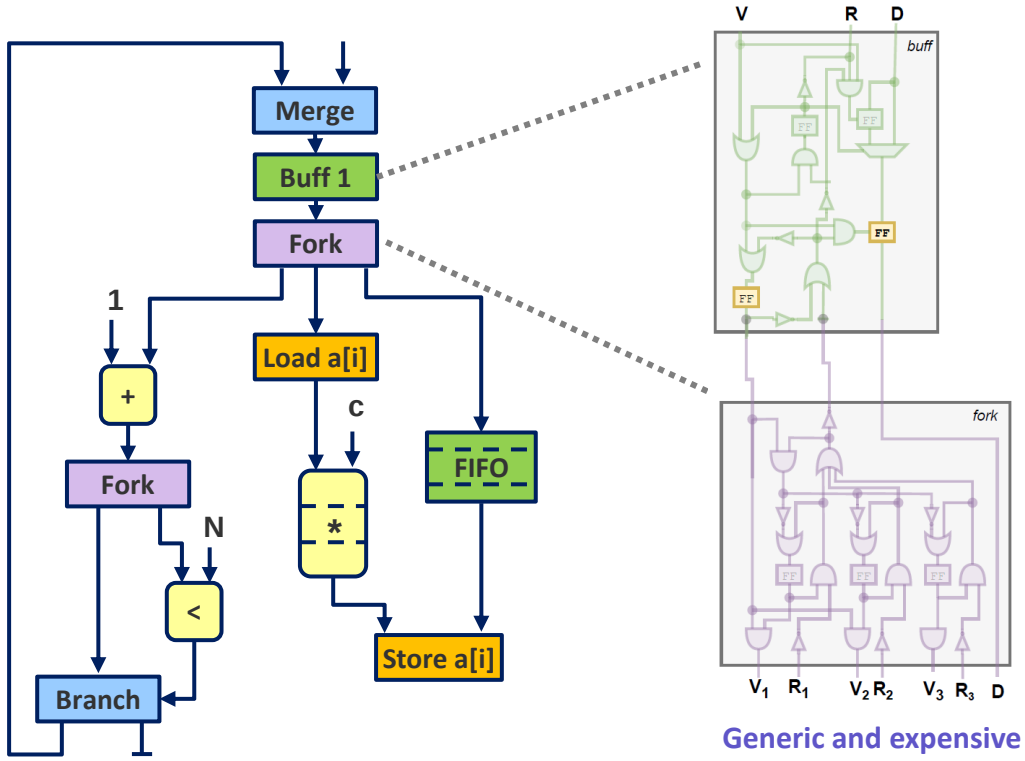
HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

Functional verification of HLS circuits using hardware simulation → inefficient and limited



Generic and expensive dataflow logic

```
for (i=0; i<N; i++) {
   a[i] = a[i]*c;
}
```


Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. FPGA 2023.
Xu and Josipović. Automatic inductive invariant generation for scalable dataflow circuit verification. ICCAD 2023.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

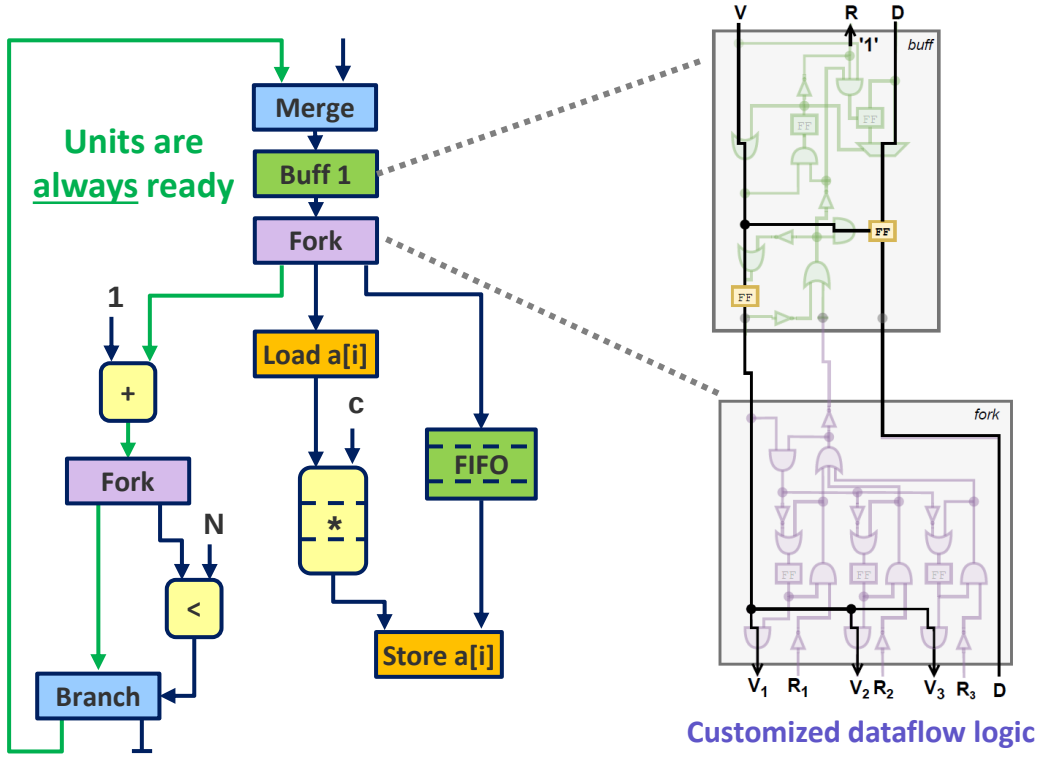
HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

Functional verification of HLS circuits using hardware simulation → inefficient and limited



Customized dataflow logic (50% area reduction)

```
for (i=0; i<N; i++) {
   a[i] = a[i]*c;
}
```


Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. FPGA 2023.
Xu and Josipović. Automatic inductive invariant generation for scalable dataflow circuit verification. ICCAD 2023.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

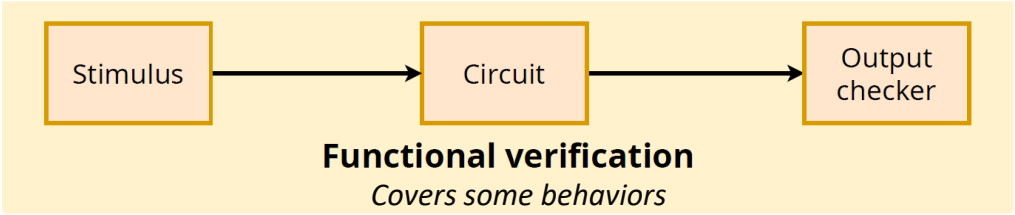
HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

Functional verification of HLS circuits using hardware simulation → inefficient and limited



Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. FPGA 2023.
Xu and Josipović. Automatic inductive invariant generation for scalable dataflow circuit verification. ICCAD 2023.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

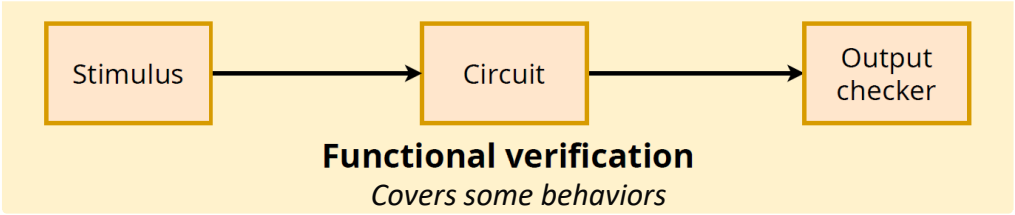
HLS often fails in extracting parallelism from software code

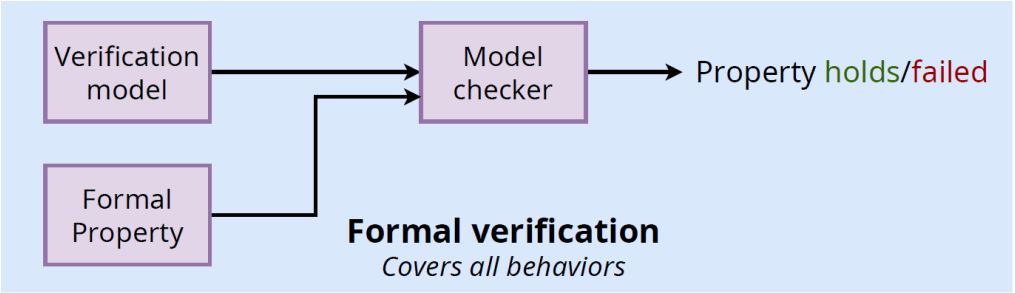
HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

Functional verification of HLS circuits using hardware simulation → inefficient and limited





A formal verification framework for improving the quality of circuits generated from software code

Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. FPGA 2023.
Xu and Josipović. Automatic inductive invariant generation for scalable dataflow circuit verification. ICCAD 2023.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

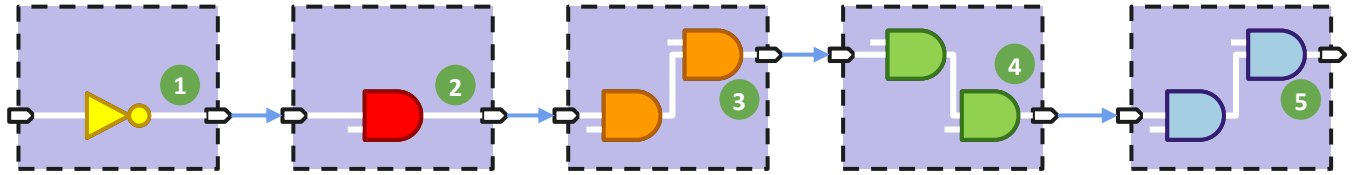
HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

Standard pipelining (register placement) is **unaware of circuit transformations** during logic synthesis and technology mapping

Standard pipelining (target: 2 logic levels after 3-LUT mapping)



Rizzi, Guerrieri, and Josipović. An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits. DAC 2023
Wang, Rizzi, and Josipović. MapBuf: Simultaneous Technology Mapping and Buffer Insertion for HLS Performance Optimization. ICCAD 2023.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

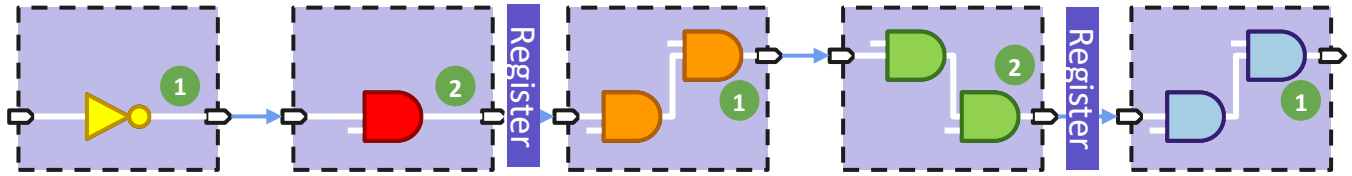
HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

Standard pipelining (register placement) is **unaware of circuit transformations** during logic synthesis and technology mapping

Standard pipelining (target: 2 logic levels after 3-LUT mapping)



Rizzi, Guerrieri, and Josipović. An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits. DAC 2023
Wang, Rizzi, and Josipović. MapBuf: Simultaneous Technology Mapping and Buffer Insertion for HLS Performance Optimization. ICCAD 2023.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

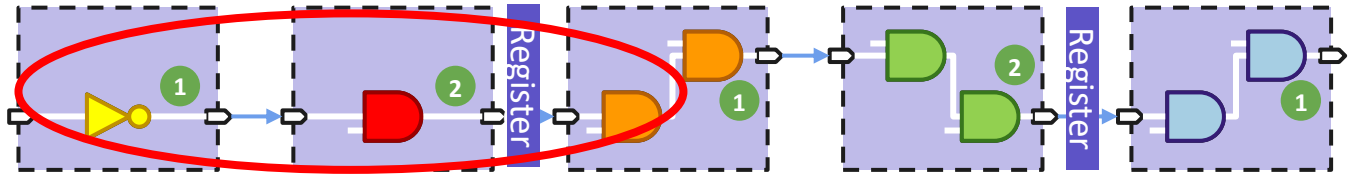
HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

Standard pipelining (register placement) is **unaware of circuit transformations** during logic synthesis and technology mapping

Standard pipelining (target: 2 logic levels after 3-LUT mapping)



Single logic level after logic synthesis → redundant regs, high latency, low frequency

Rizzi, Guerrieri, and Josipović. An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits. DAC 2023
Wang, Rizzi, and Josipović. MapBuf: Simultaneous Technology Mapping and Buffer Insertion for HLS Performance Optimization. ICCAD 2023.

Bridging the Gap Between Software and Hardware

SW

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

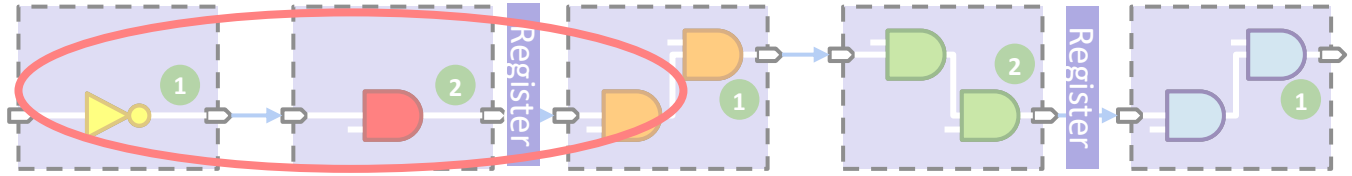
HLS circuits need hardware-level functional verification

It is difficult for HLS to account for reconfigurable platform details

HW

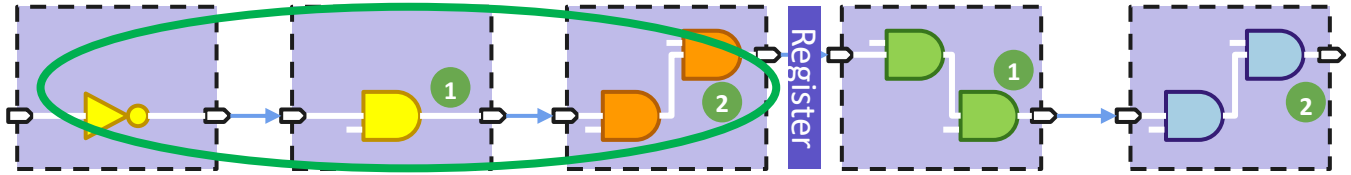
Standard pipelining (register placement) is **unaware of circuit transformations** during logic synthesis and technology mapping

Standard pipelining (target: 2 logic levels after 3-LUT mapping)



Single logic level after logic synthesis → redundant regs, high latency, low frequency

Simultaneous pipelining & technology mapping (our work)



Fewer registers, low latency, high frequency

Implementation-aware compiler optimizations for fast and small circuits

Rizzi, Guerrieri, and Josipović. An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits. DAC 2023
Wang, Rizzi, and Josipović. MapBuf: Simultaneous Technology Mapping and Buffer Insertion for HLS Performance Optimization. ICCAD 2023.

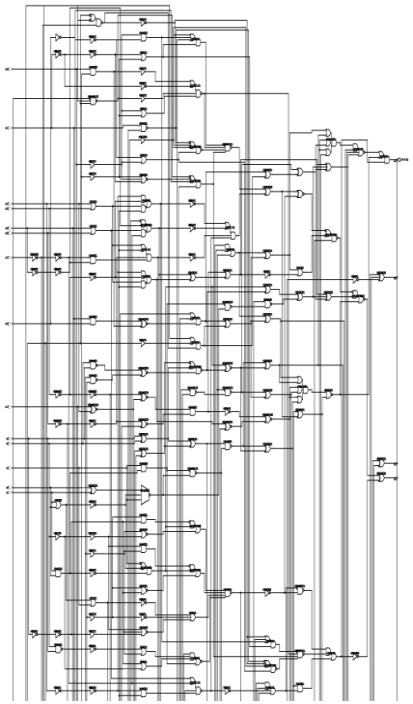
Bridging the Gap Between Software and Hardware

High-level abstractions
programming languages,
software applications

HLS compilers
formal methods,
electronic design automation

Hardware accelerators
systems, digital design,
computer architecture

```
#define PI 3.14159265358979323846

complex* DFT_naive(complex* x, int
  complex* X = (complex*) malloc(s
  int k, n;
  for(k = 0; k < N; k++) {
    X[k].re = 0.0;
    X[k].im = 0.0;
    for(n = 0; n < N; n++) {
      X[k] = add(X[k], multiply(x[
                                co


    }
  }

  return X;
}
```



Enable diverse users to accelerate compute-intensive applications on hardware platforms

Thanks! ☺

Research group:

dynamo.ethz.ch

Dynamatic HLS tool:

dynamatic.epfl.ch

Dynamatic 2.0 coming soon!