# Advanced Computer Architecture

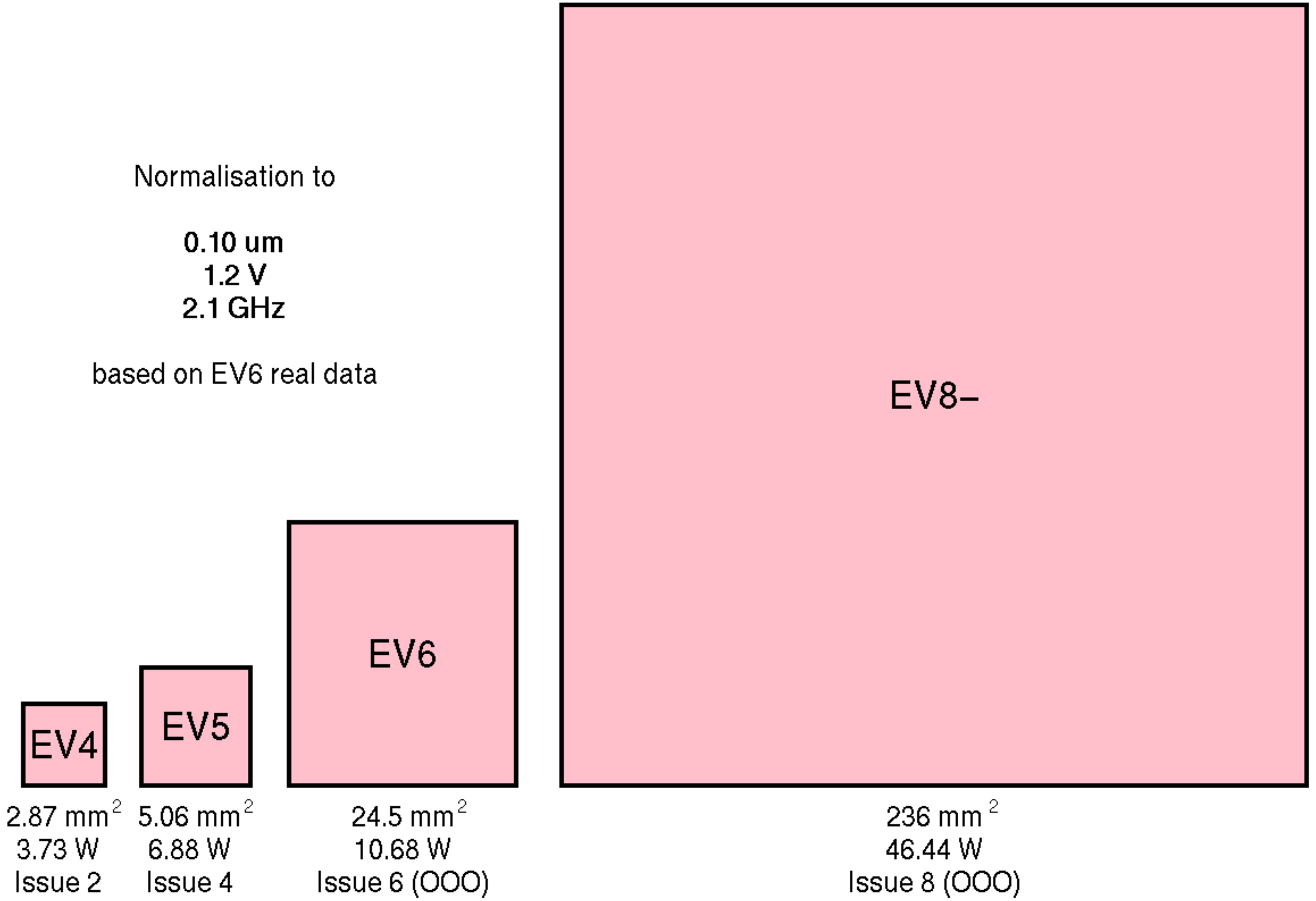
—

## Part II: Embedded Computing From Processor Customization to HLS

Paolo Ienne

<paolo.ienne@epfl.ch>

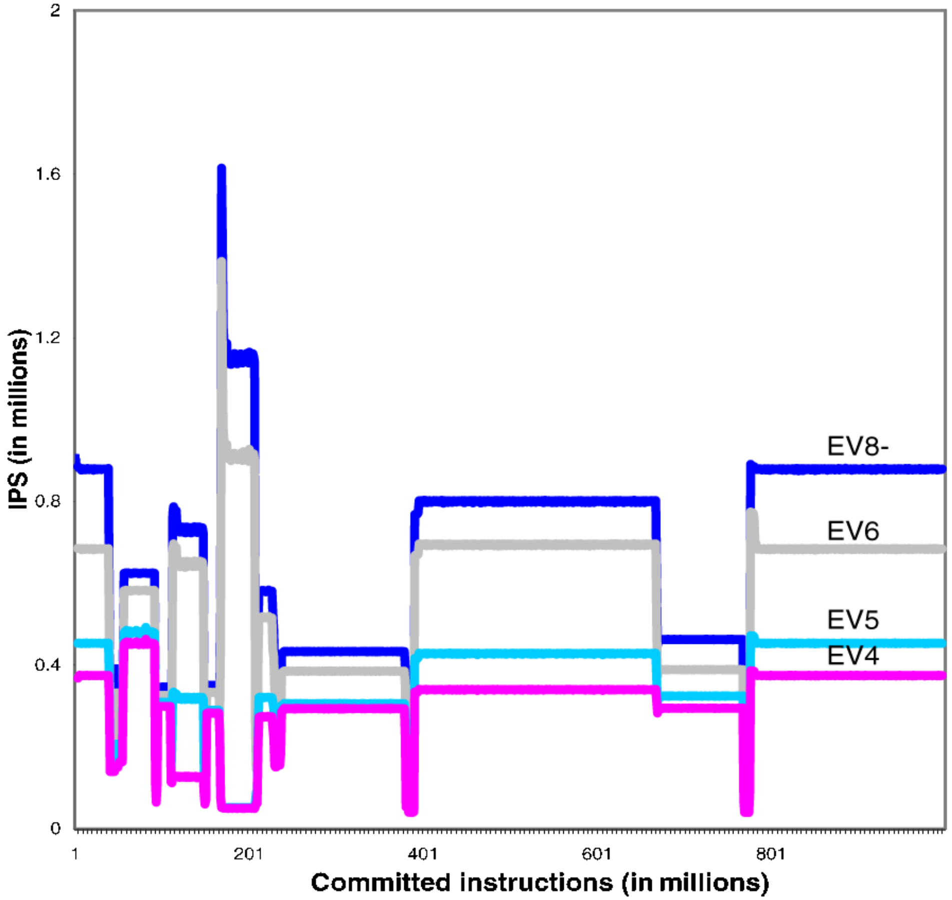
# Four Generations of Alpha



Source: Kumar et al., MICRO-36, © IEEE 2003

# Cost of Additional Performance

- **Very** roughly (at constant technology):
  - 80x area
  - 12x power
  - 2-3x performance
- **Pervasive mobile applications** cannot afford such costs:
  - Volume products are still sensitive to **area**
  - **Energy** is at a premium!



Source: Kumar et al., MICRO-36, © IEEE 2003

**Can we get the performance without paying the price?!**

# Reminder of Embedded Processors Specificities

- Cost used to be the only concern; now **performance/cost is at premium** and still not performance alone as in PCs (Intel model); performance is often a constraint
- **Binary compatibility** is less of an issue for embedded systems
- Systems-on-Chip make **processor volume irrelevant** (moderate motivation toward single processor for all products)

**DSPs?**

# Increasing the Efficiency of Implementations

**From C programs to more efficient "programmable" solutions**

- **Automatic Processor Customization**
  1. ISA configuration and extension from applications
     - The "fourth generation HLS" ?! (see Martin & Smith, IEEE DTC 2009)

- **High-Level Synthesis**
  2. Statically scheduled HLS
     - The "VLIW" way…
     - Taming DSP and multimedia applications

  3. Dynamically scheduled HLS
     - Conquering prediction and speculation
     - A better match to control-dominated irregular applications?

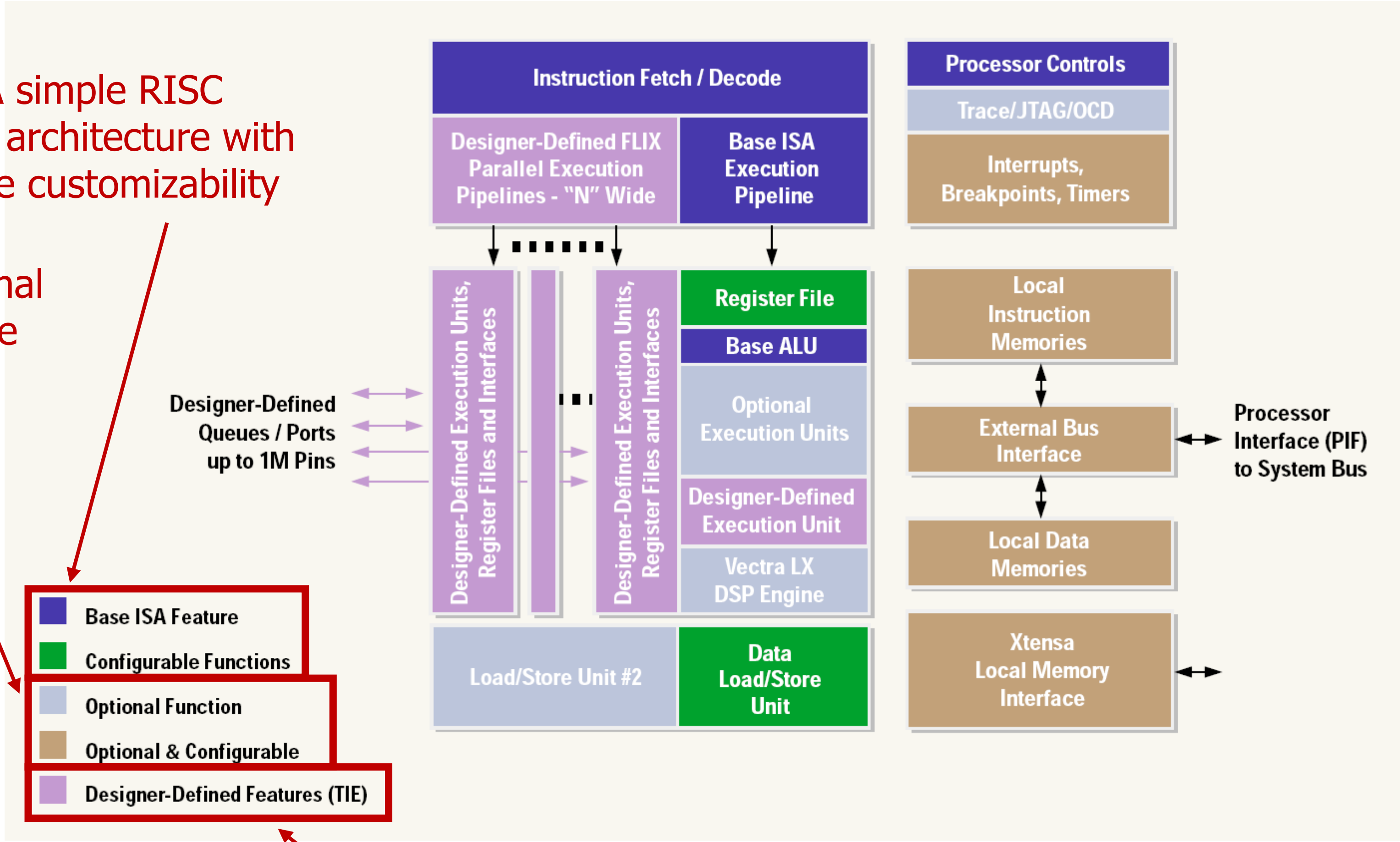
# 1

Automatic Processor Customization

# Tensilica Xtensa

A simple RISC basic architecture with some customizability

Several optional customizable predefined blocks


Instruction Fetch / Decode
Designer-Defined FLIX Parallel Execution Pipelines - "N" Wide
Base ISA Execution Pipeline
Processor Controls
Trace/JTAG/OCD
Interrupts, Breakpoints, Timers
Designer-Defined Queues / Ports up to 1M Pins
Designer-Defined Execution Units, Register Files and Interfaces
Designer-Defined Execution Units, Register Files and Interfaces
Register File
Base ALU
Optional Execution Units
Designer-Defined Execution Unit
Vectra LX DSP Engine
Local Instruction Memories
External Bus Interface
Processor Interface (PIF) to System Bus
Local Data Memories
Load/Store Unit #2
Data Load/Store Unit
Xtensa Local Memory Interface
Base ISA Feature
Configurable Functions
Optional Function
Optional & Configurable
Designer-Defined Features (TIE)

Prepared interfaces for user-defined customizations

© Tensilica 2006

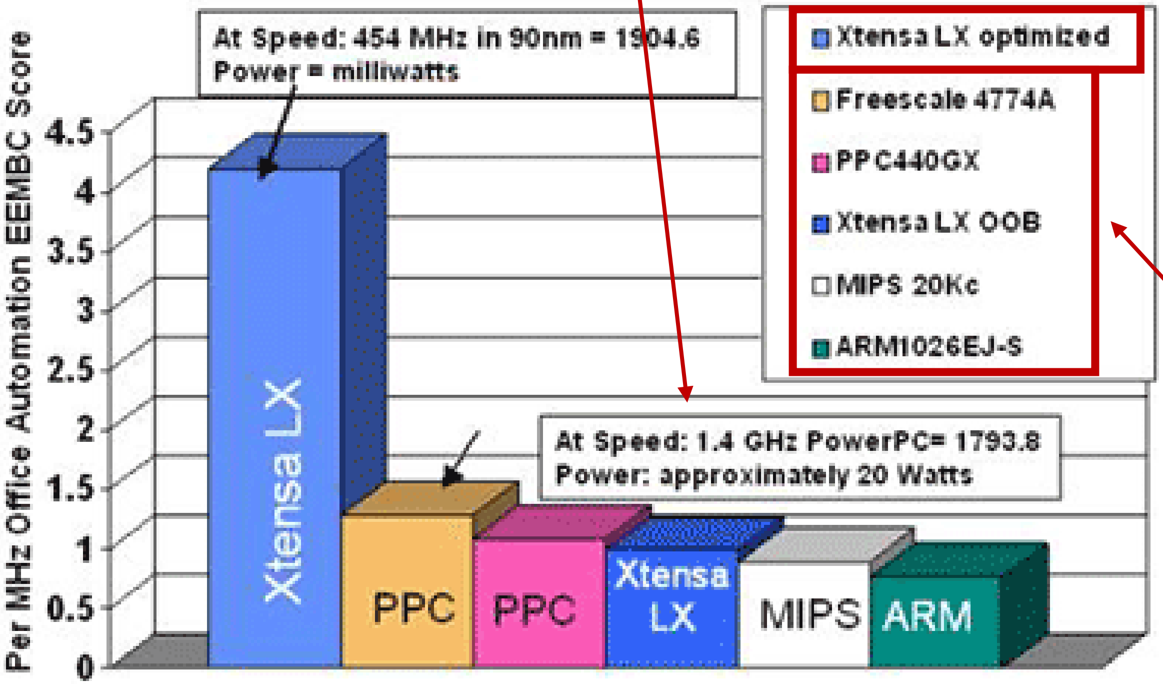
# ARC 625D
# Configuration Possibilities

- Processor:
  - Register file type and size
  - Number of interrupts and pins
  - Reset state
  - Endianness
- Cache:
  - Cache type: Instruction and/or Data
  - Size: 2k - 32k Bytes
  - Ways: 1 - 4
  - Line Length: 16 - 128 bytes
- Closely Coupled Memory:
  - Instruction RAM: 1k - 512k bytes
  - Data RAM: 2k - 32k bytes
- Instructions:
  - NORM - find the first "0" in a 32 bit word
  - SWAP - switch locations of the top and bottom 16 bit fields
  - MULT32 - fast 32 x 32 bit multiplier
- DSP Functions:
  - 24x24 MAC
  - Dual 16x16 MAC
  - 32x16 MAC
  - Extended Arithmetic Package
  - Dual Viterbi Butterfly
  - CRC Acceleration
  - Audio Acceleration Package
  - XY Memory 1-2 Banks, 1k - 32k bytes, single or dual ported
- Peripherals:
  - Timers
- Bus Components:
  - BVCI Arbiter
  - AMBA AHB
- Debug:
  - JTAG interface
  - Actionpoints

# Benefits of Customization

Area and power efficiency

Customization quickly raises the same architecture and breaks out of the pack



Out-of-the-Box Xtensa LX is in the pack with other processors

**Beware**: This is a marketing slide…

© Tensilica 2006

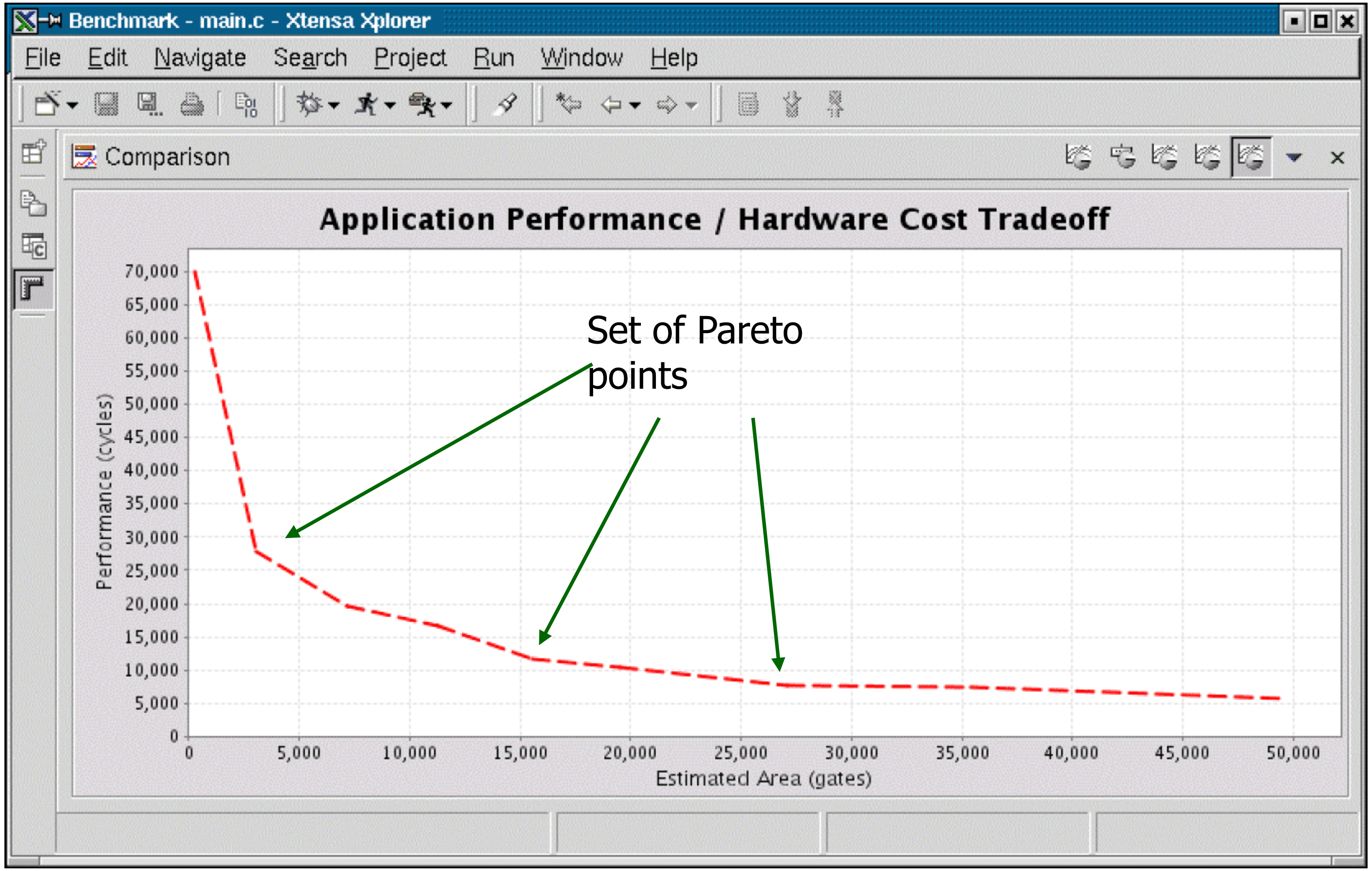
# Mainstream ASIC/FPGA Processors and Specialisation?

All the recent embedded ASIC/FPGA processors offer some sort of specialisation:

- **Parametric resources** (STM Lx/ST200, ARC Cores, Tensilica Xtensa, Altera Nios, etc.)
- **Arbitrary functional units or tightly coupled coprocessors** (STM Lx/ST200, IFX Carmel 20xx, ARM, Tensilica Xtensa, Altera Nios, MIPS CorExtend, etc.)

But all assume an onerous
**manual study and design!**

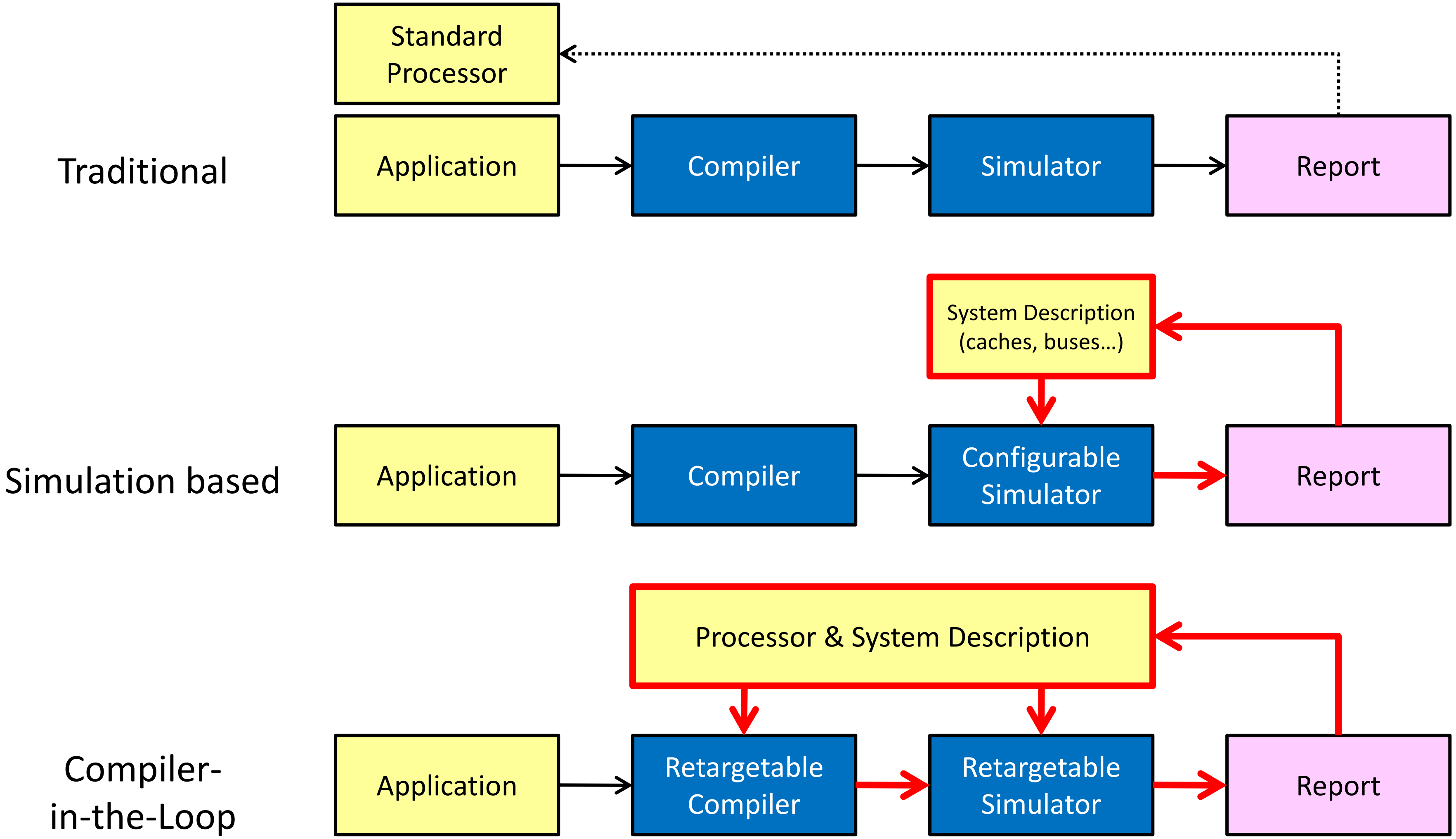
# Tensilica Xpres
# Automatic Configuration Tool



Set of Pareto points

Interesting CS problems to explore the design space efficiently!

# Compilers as Design Tools



# EDA Meets Computer Architecture

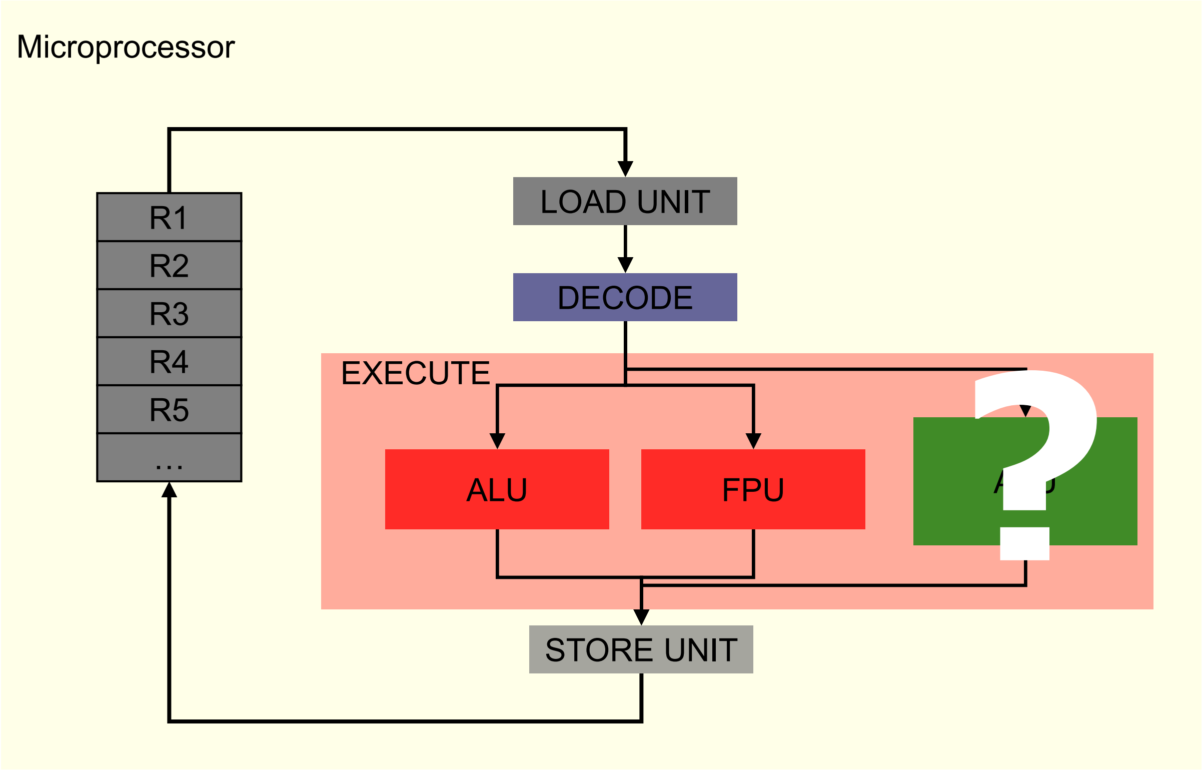
- Tensilica (founded 1997) has been bought by **Cadence** in 2013

- ARC International (founded in the early 1990s) has been bought by Virage Logic in 2009 and Virage Logic has been acquired by **Synopsys** in 2010

- Little known progress in **automating** the customization process, though…

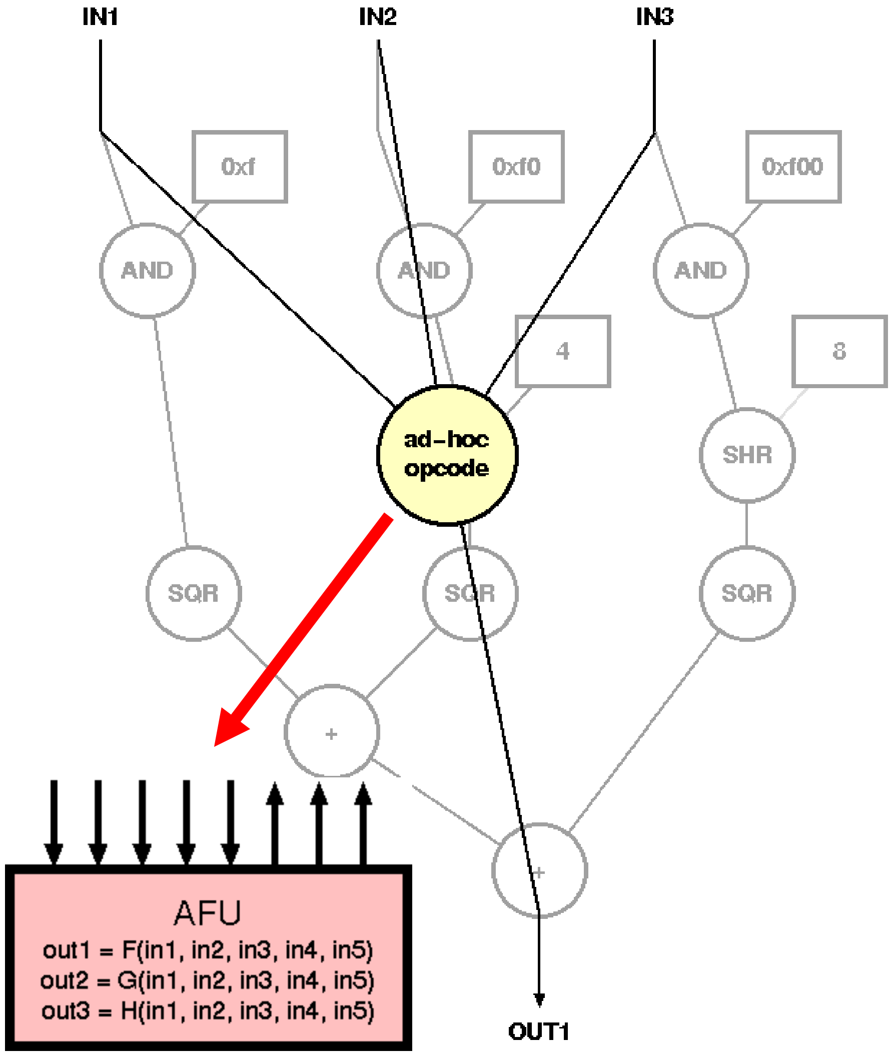
# User Designed Customizations? Instruction Set Extensions

- A “safe” technique for extensive customization



- Available in many commercial processors (from MIPS, STM, IFX, Tensilica, ARC, Xilinx, Altera,…)

# Instruction Set Extensions (ISE)


IN1
IN2
IN3
0xf
0xf0
0xf00
AND
AND
AND
4
8
ad-hoc opcode
SHR
SQR
SQR
SQR
+
+
OUT1
AFU
out1 = F(in1, in2, in3, in4, in5)
out2 = G(in1, in2, in3, in4, in5)
out3 = H(in1, in2, in3, in4, in5)

- Collapse a subset of the Direct Acyclic Graph nodes into a single Functional Unit (AFU)
  - Exploit cheaply the parallelism within the basic block
  - Simplify operations with constant operands
  - Optimise sequences of instructions (logic, arithmetic, etc.)
  - Exploit limited precision

# Elementary Motivational Example
# An Important Kernel…

```
/* init */
a <<= 8;
/* loop */
for (i = 0; i < 8; i++) {
    if (a & 0x8000) {
        a = (a << 1) + b;
    } else {
        a <<= 1;
    }
}
return a & 0xffff;
```

Shift-and-add
unsigned
8 x 8-bit
multiplication

# Software Predication

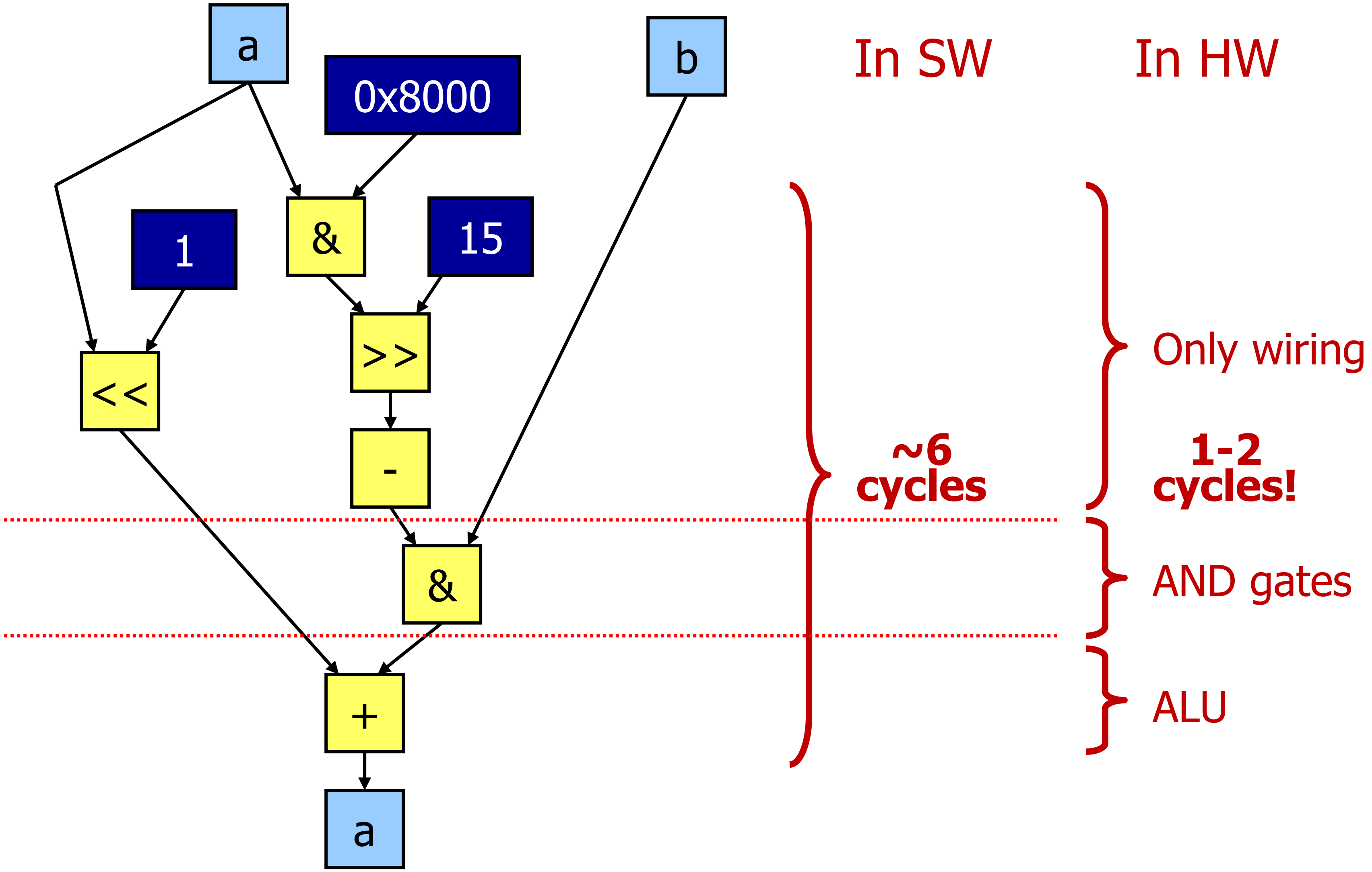
```
/* init */
a <<= 8;
/* loop */
for (i = 0; i < 8; i++) {
   p1 = - ((a & 0x8000) >> 15);
   a = (a << 1) + b & p1;
}
return a & 0xffff;
```

Shift

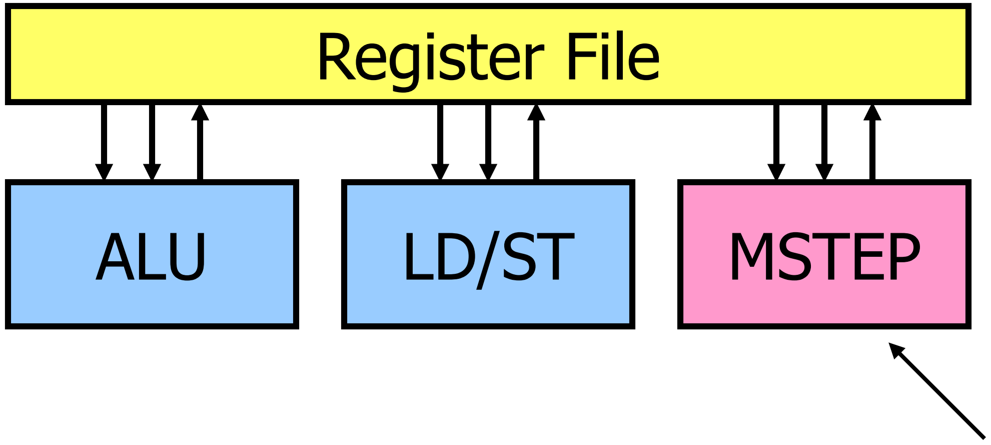
Predicated Add

Predicate mask
(0 or –1 = 0xfffffff)

# Loop Kernel DAG



# New Unit To Accelerate Shift-and-Add Multiplication Loop



```
if (Rn [31] = = 1)
then Rn ← (Rn << 1) + Rm
else Rn ← (Rn << 1)
```

One instruction added

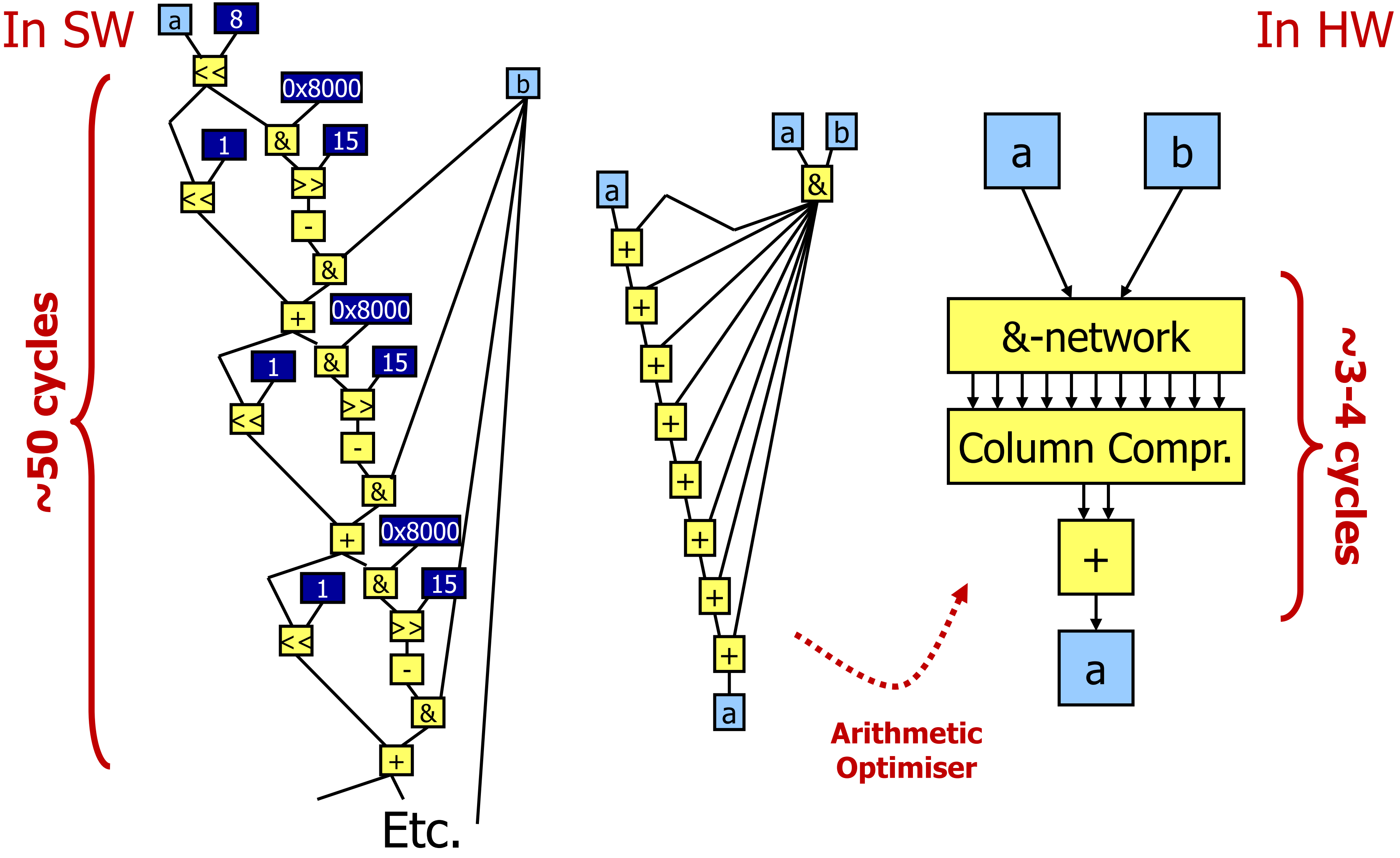
➔

loop kernel reduced to **15-30%**

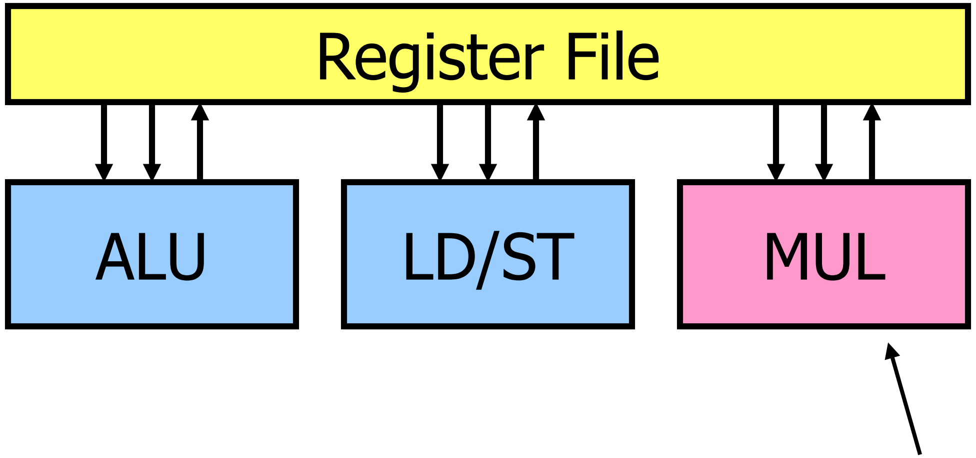
# Loop Unrolling

```
/* init */
a <<= 8;
/* no loop anymore */
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
return a & 0xffff;
```

# Full DAG


In SW
~50 cycles
a
8
<<
0x8000
b
&
15
1
<<
>>
-
&
+
0x8000
1
&
15
<<
>>
-
&
+
0x8000
1
&
15
<<
>>
-
&
+
Etc.
a
b
a
&
+
+
+
+
+
+
+
+
a
Arithmetic Optimiser
In HW
a
b
&-network
Column Compr.
+
a
~3-4 cycles

# New Unit To Accelerate Multiplication?! Yeah, a MUL…



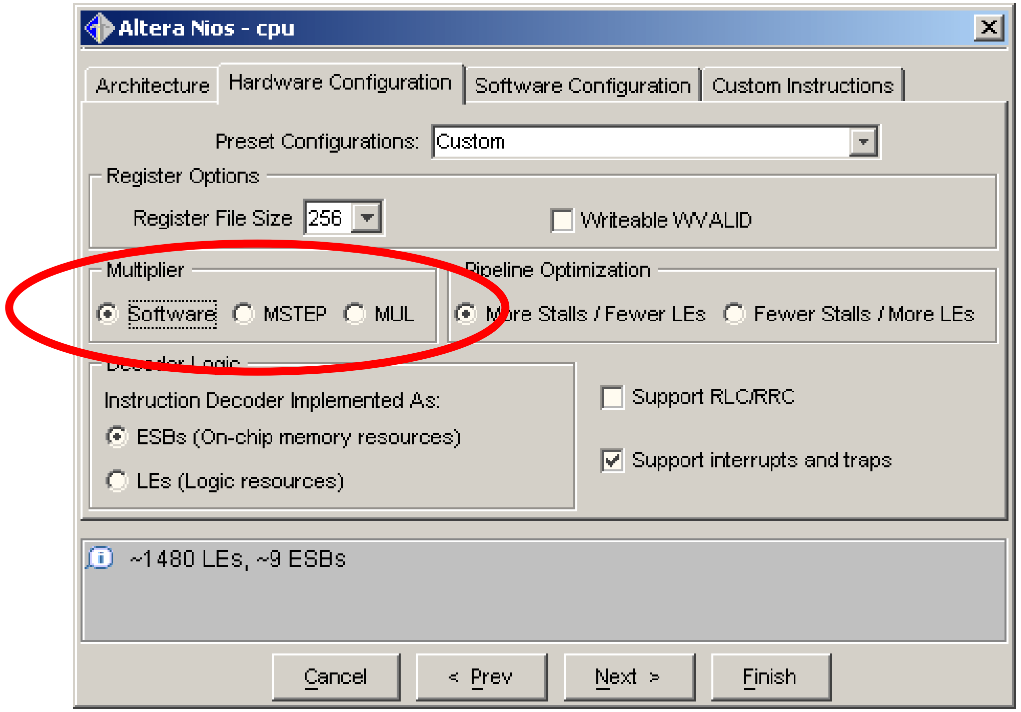
R*n* ← (R*n* & 0x0000.ffff) x (R*m* & 0x0000.ffff)

One instruction added

➔

function reduced by a factor **10-15**

# Classic "Specialisation"…



Altera Nios - cpu

Architecture | Hardware Configuration | Software Configuration | Custom Instructions

Preset Configurations: Custom

Register Options

Register File Size: 256

□ Writeable WVALID

Multiplier: ● Software ○ MSTEP ○ MUL

Pipeline Optimization: ● More Stalls / Fewer LEs ○ Fewer Stalls / More LEs

Decoder Logic

Instruction Decoder Implemented As:

● ESBs (On-chip memory resources)

○ LEs (Logic resources)

□ Support RLC/RRC

☑ Support interrupts and traps

~1480 LEs, ~9 ESBs

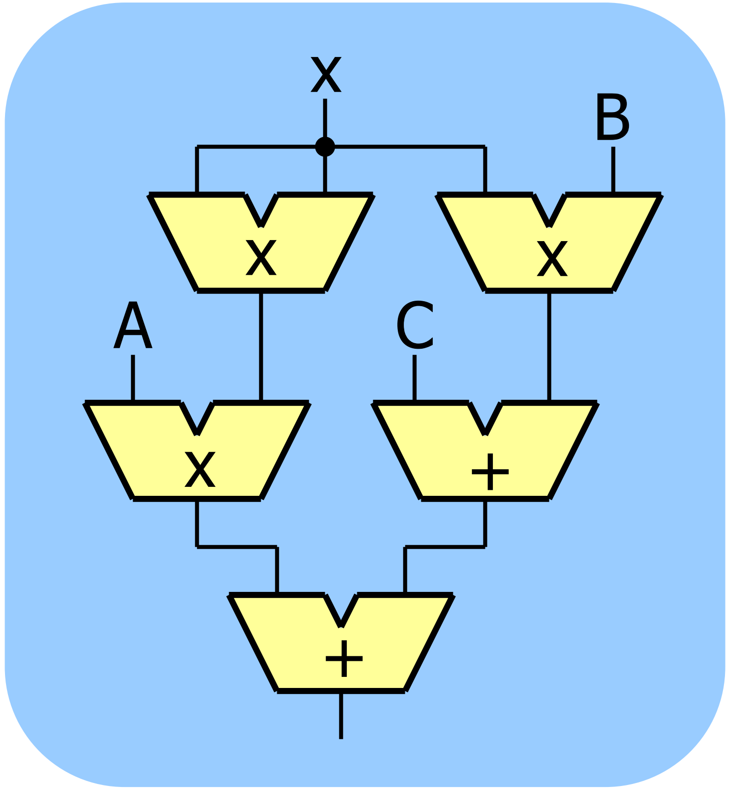
© Altera 2003

# Why Hardware Is Better?

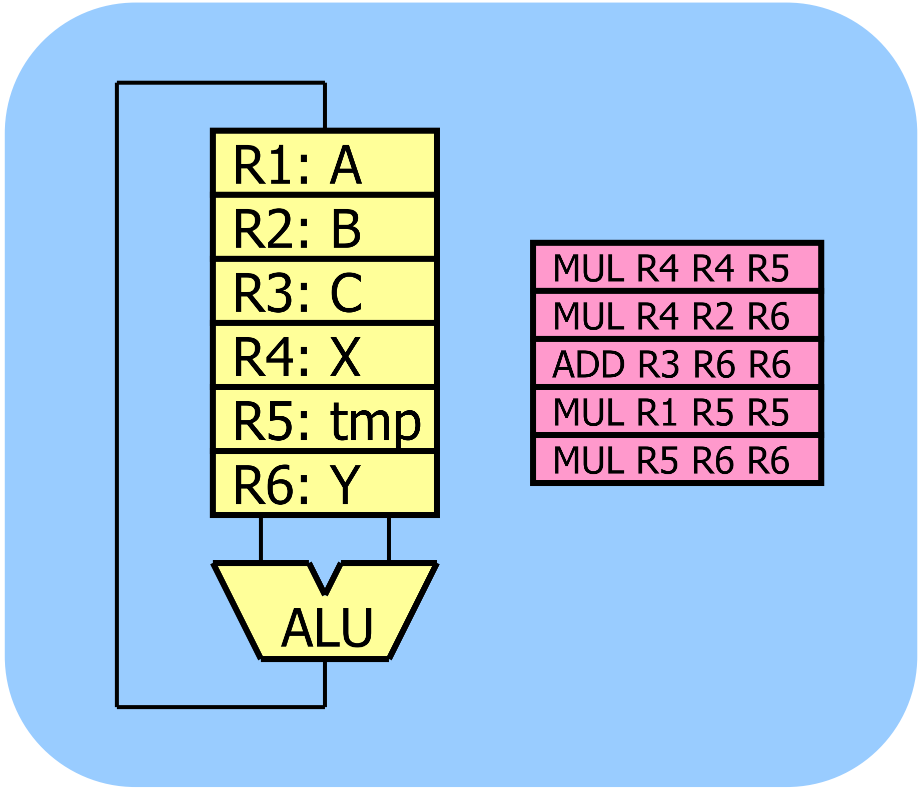
- Spatial computation
  - Cheap “ILP” without true ILP support
- No quantization of time in clock cycles for each operation/instruction
  - Operation chaining
- Hardware is different
  - Constants may be propagated
  - Precision can be tuned (bitwidth analysis)
  - Arithmetic components can be optimized

# Spatial Computation

Spatial Computing



Temporal Computing

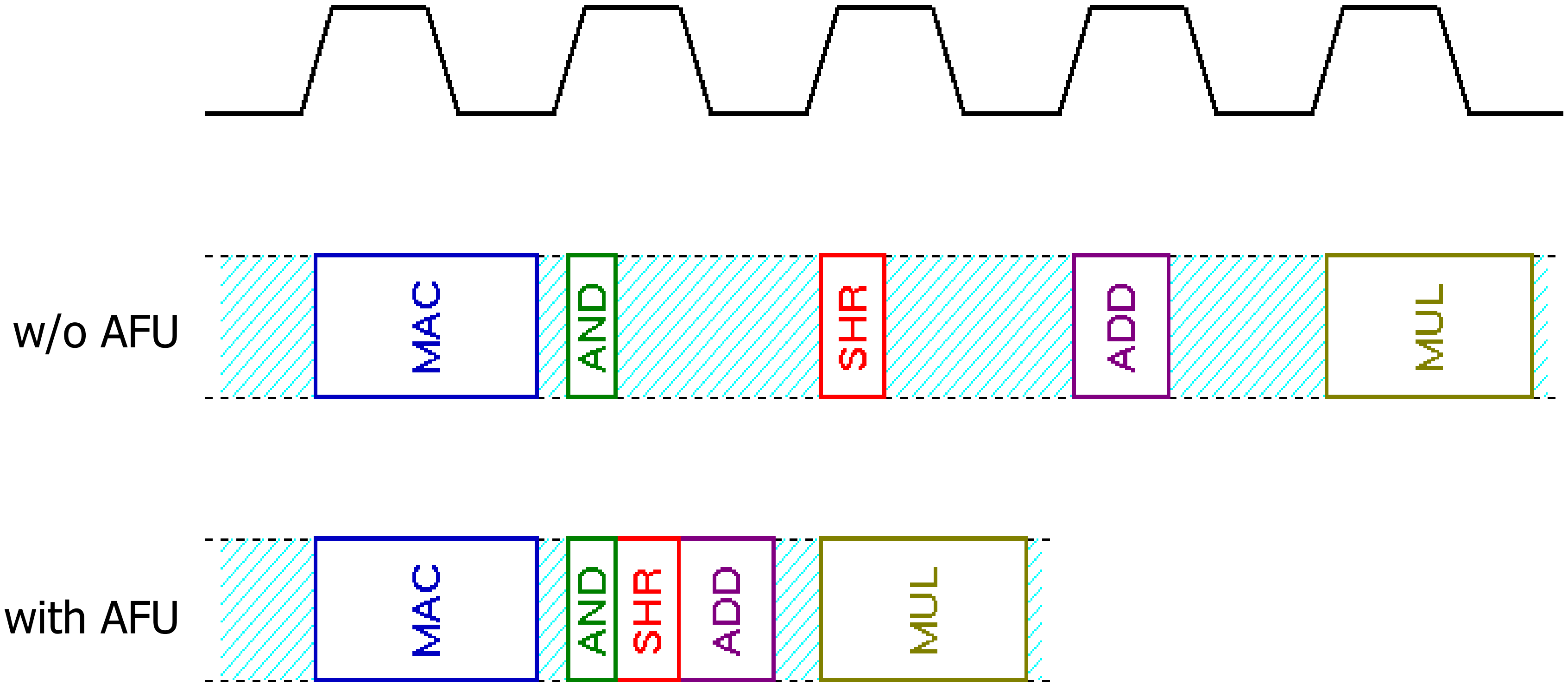

R1: A
R2: B
R3: C
R4: X
R5: tmp
R6: Y

```
MUL R4 R4 R5
MUL R4 R2 R6
ADD R3 R6 R6
MUL R1 R5 R5
MUL R5 R6 R6
```

Adapted from DeHon, © ACM 1999

# No Time Quantization

- Effective occupation of the execute stage



# Constants Example

```
/* an excerpt from adpcm.c */
/* adpcmdecoder, mediabench */

vpdiff = step>>3;
if (delta & 4) vpdiff += step;
if (delta & 2) vpdiff += step>>1;
if (delta & 1) vpdiff += step>>2;
```

- Exploited to reduce complexity—e.g.,

  `a*5 → a<<2+a`

- Hardcoded into logic

- Bitwise operations (e.g., on `delta`, `step`) reduce to wires

# Bitwidth Analysis Example

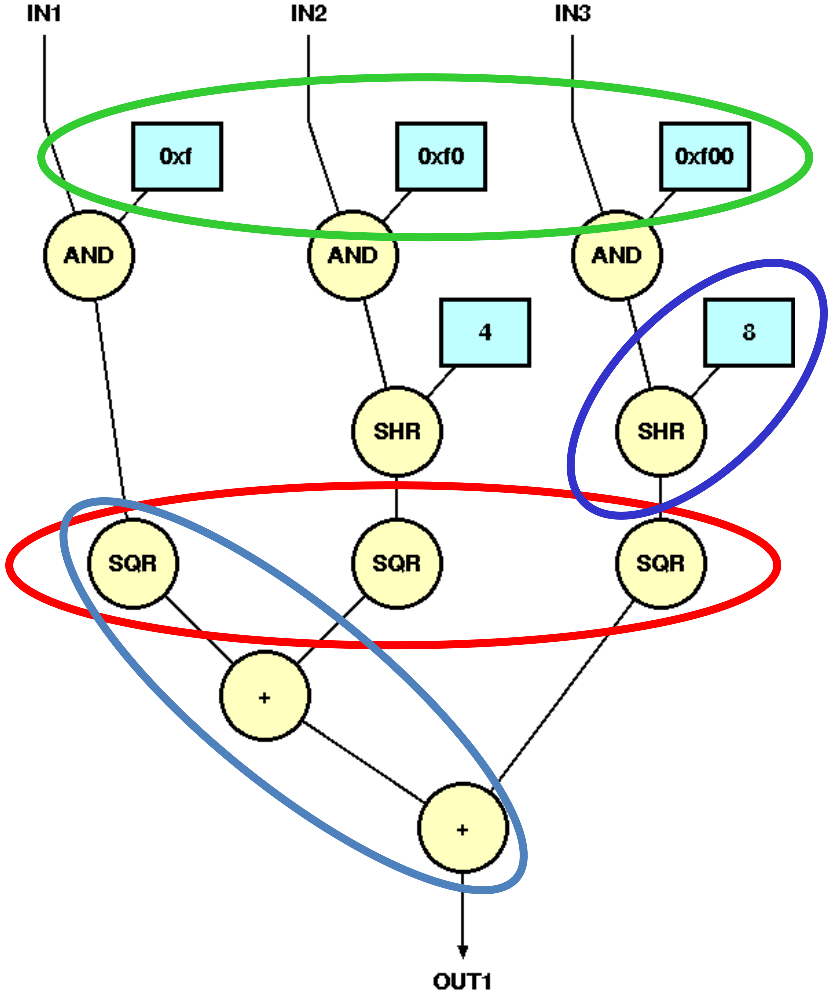
```
/* an excerpt from adpcm.c */
/* adpcmencoder, mediabench */

index = indexTable[delta];

if (index < 0) index = 0;
if (index > 88) index = 88;

step = stepsizeTable[index];
```

- $0 \leq$ `index` $\leq 88$
- 7 bits sufficient for representation
- Faster arithmetic components, etc.

# Arithmetic Optimizations

- Arithmetic operations often appear in groups (dataflow graphs)
- A literal/sequential implementation may not make the best of the potential available
- A different number representation can be a **game-changer**
  - May bring large advantages, often without higher hardware cost
  - Big O complexity O() may change with a different representation!
  - E.g., carry-save adders, column compressors, etc.
- Typical example: MAC
  - Only marginally slower than corresponding MUL
  - Practically same complexity

# Why Hardware Is Better?



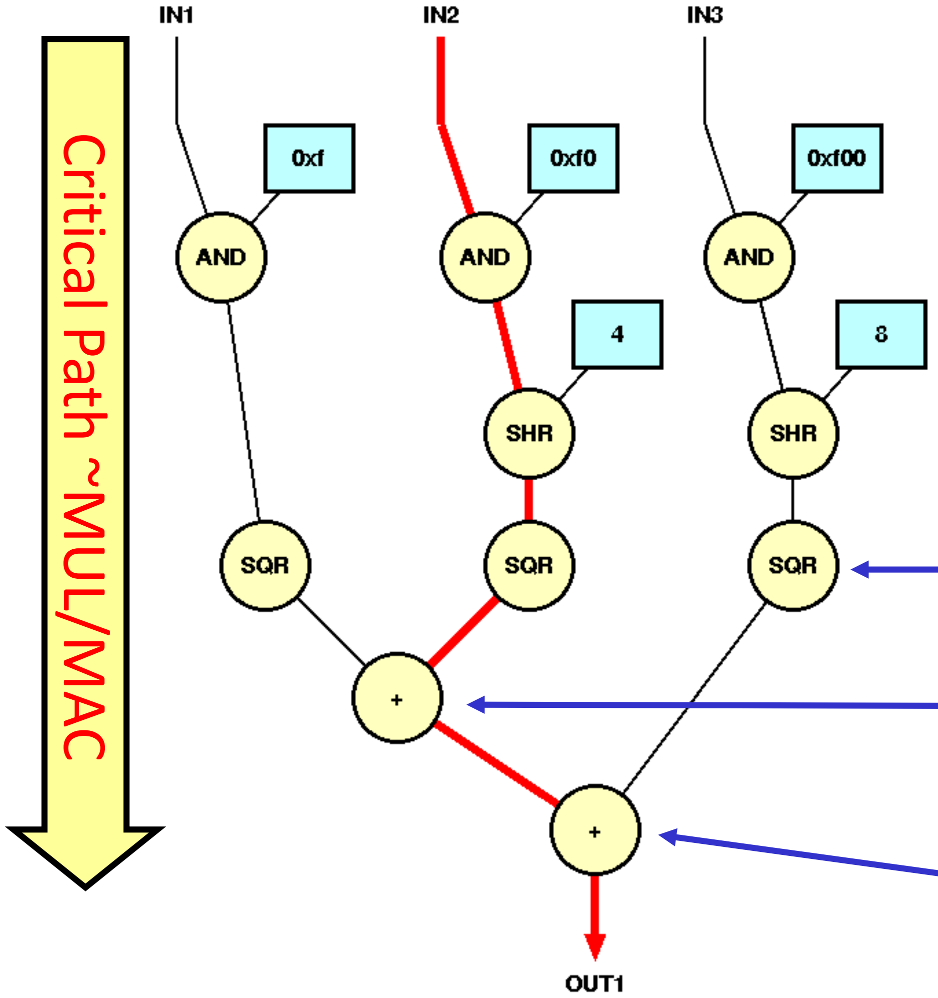
Exploit constant for logic simplification

Some operations reduce to wires in hardware

Exploit data parallelism in hardware

Exploit arithmetic properties for efficient chaining of arithmetic operations (e.g., carry save)

# Gain Potentials in Ad-hoc FUs: Tangible Cycle Savings Possible



Critical Path ~MUL/MAC

No hardware needed

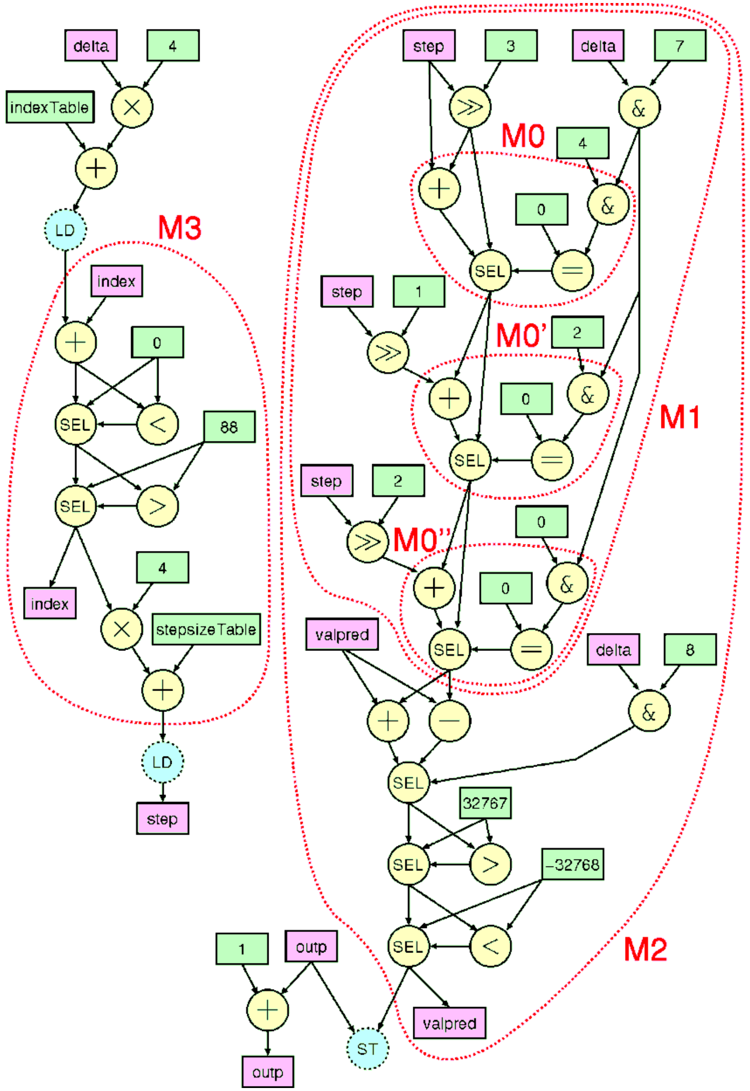
No hardware needed

Similar to MUL but less additive terms

One more additive term in carry-save/Wallace tree

One more additive term in carry-save/Wallace tree (as in MAC)

# Automatic ISE Discovery

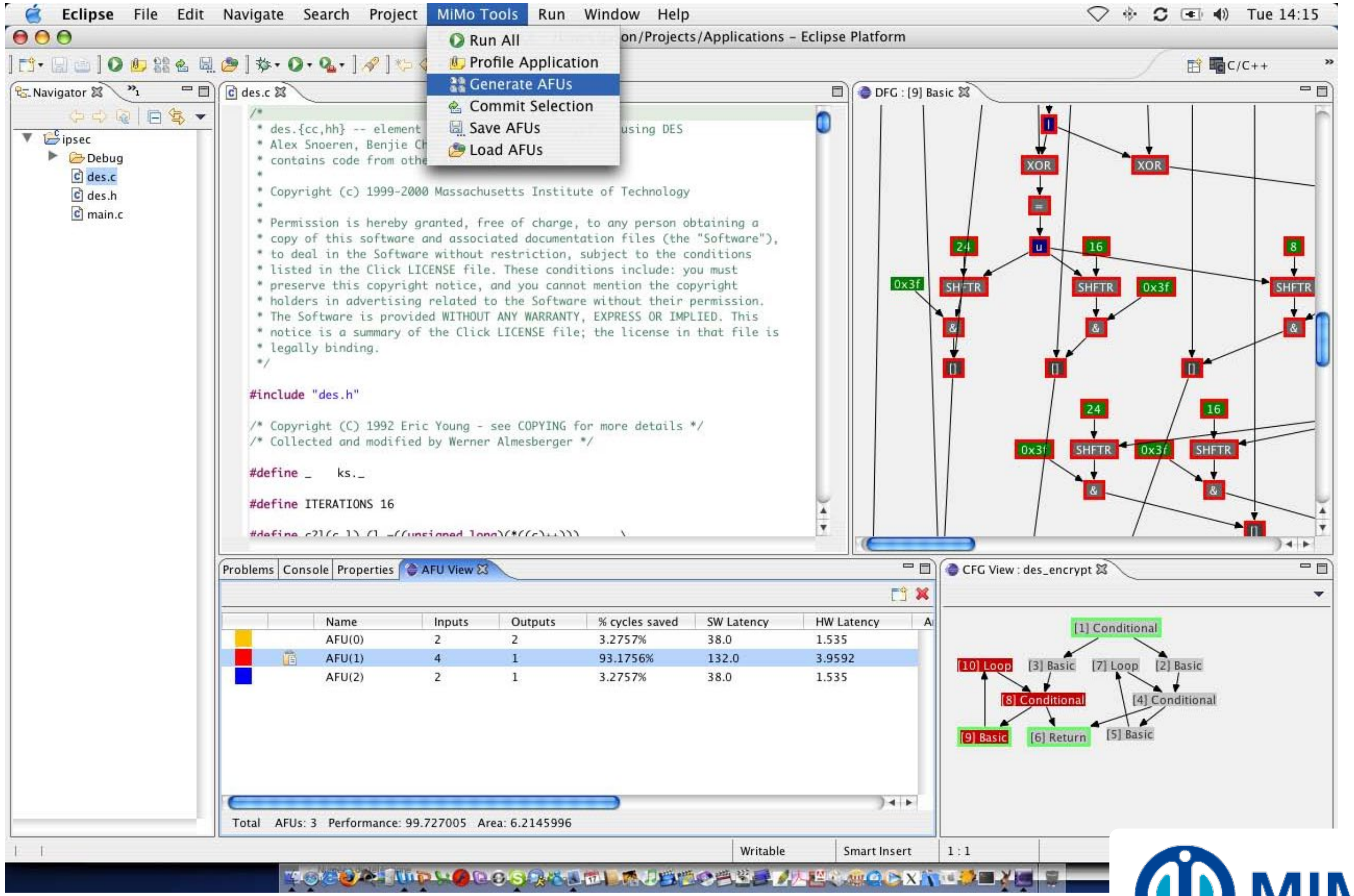


**Formulate it as an optimization problem**

Find subgraphs

1. having a user-defined maximum number of inputs and outputs,
2. convex,
3. possibly including disconnected components, and
4. that maximise the overall speedup

# Automatic ISE Discovery



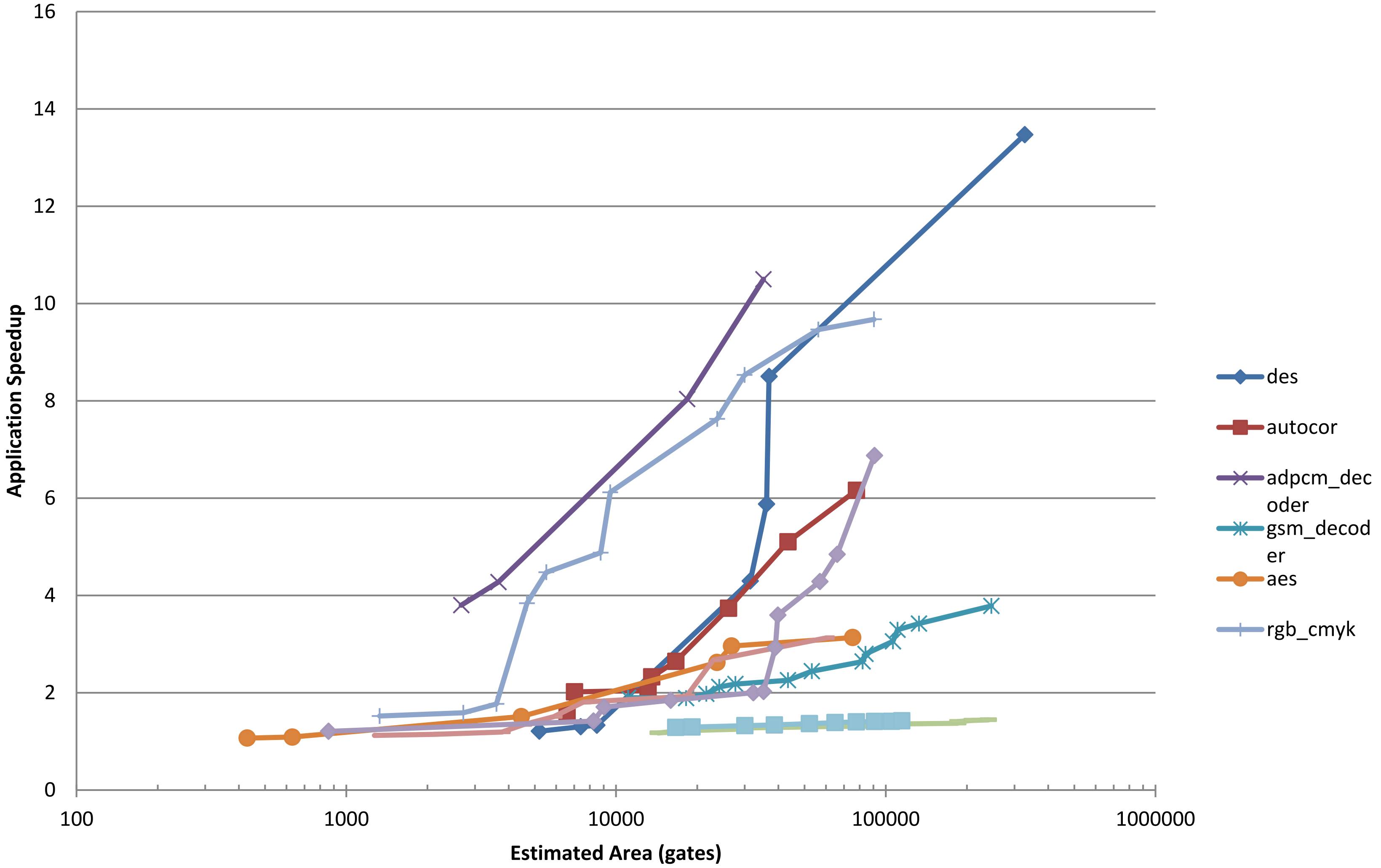
```
/*
 * des.{cc,hh} -- element                                  using DES
 * Alex Snoeren, Benjie Ch
 * contains code from othe
 *
 * Copyright (c) 1999-2000 Massachusetts Institute of Technology
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the "Software"),
 * to deal in the Software without restriction, subject to the conditions
 * listed in the Click LICENSE file. These conditions include: you must
 * preserve this copyright notice, and you cannot mention the copyright
 * holders in advertising related to the Software without their permission.
 * The Software is provided WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED. This
 * notice is a summary of the Click LICENSE file; the license in that file is
 * legally binding.
 */

#include "des.h"

/* Copyright (C) 1992 Eric Young - see COPYING for more details */
/* Collected and modified by Werner Almesberger */

#define _    ks._

#define ITERATIONS 16

#define c2l(c,l) (l =((unsigned long)(*((c)++)))      \
```

| | | Name | Inputs | Outputs | % cycles saved | SW Latency | HW Latency | A |
|---|---|---|---|---|---|---|---|---|
| 🟨 | | AFU(0) | 2 | 2 | 3.2757% | 38.0 | 1.535 | |
| 🟥 | | AFU(1) | 4 | 1 | 93.1756% | 132.0 | 3.9592 | |
| 🟦 | | AFU(2) | 2 | 1 | 3.2757% | 38.0 | 1.535 | |

Total AFUs: 3 Performance: 99.727005 Area: 6.2145996

MIMOSYS

# Automatic ISE Discovery Examples



# Processor Customization?

- Arguably the **most widespread method of designing embedded hardware**: selecting one of very many existing processors or configuring the parameters of a family of processors amounts to customization for a set of applications
- **Little automation**, though: still mostly a manual design-space exploration; glimpses of automation in the 2000s seem lost
- Automatic ISE discovery could be a more promising automatic customization opportunity, but also disappeared in the late 2000s (the “fourth generation HLS” is dead?)
  - Pros: Focus on automatic design of datapath and leave control to manually optimized processors (prediction, speculation, etc.)
  - Cons: Limited scope of exploitable parallelism (datapath parallelism and convertible control—e.g., predication, unrolling)

# 2

Statically Scheduled High-Level Synthesis

*(with Lana Josipović)*

# Beyond Dataflow

- Somehow, ISE is confined to dataflow or convertible control flow, and this limits exploitable parallelism

- Traditional **HLS** gets rid of the processor altogether and uses the **C/C++ specification to build hardware**

- It represents an attempt (started in the late ’80s and early ’90s) to raise the abstraction level of hardware design above the classic RTL level (i.e., synthesizable VHDL and Verilog)

# A Bit of History

- Generation 0 (1970s), prehistory
  - Groundbreaking academic work
- Generation 1 (1980s until early 1990s)
  - Mostly important academic work; few commercial players
  - Focus on scheduling, binding, etc.
  - Almost competing in adoption with RTL logic synthesis
- Generation 2 (mid 1990s until early 2000s)
  - Main EDA players offer commercial HLS tools; commercial failure
  - Assumed RTL designers would embrace the technology, but there was not enough gain for them
  - Wrong programming languages (VHDL or new languages)
- **Generation 3 (from early 2000s)**
  - Currently available commercially (e.g., Vivado HLS); some successes
  - Connected to the rise of FPGAs (fast turnaround, inexperienced designers, etc.)
  - Focus on C/C++ and on demanding dataflow/DSP applications
  - Better results (progress in compilers, including VLIW)

Adapted from Martin & Smith, IEEE DTC 2009

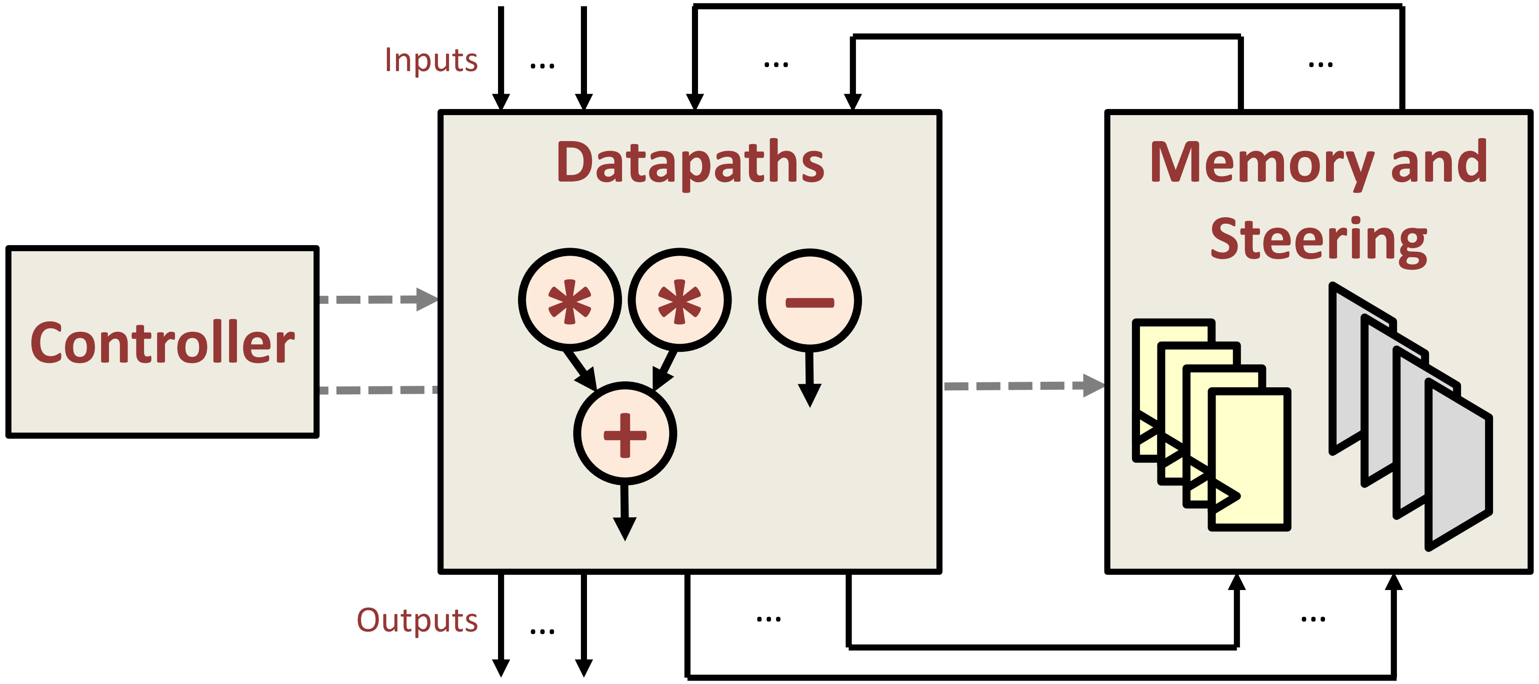
# What Circuits Do We Want HLS to Generate?

- Output of HLS is ill-defined
  - An example could be to generate always the same hardware (the RTL of a software processor) and binary code for it—hardly what we usually mean by HLS…

- The informal expectation is a circuit much more massively parallel than what a classic software processor can achieve

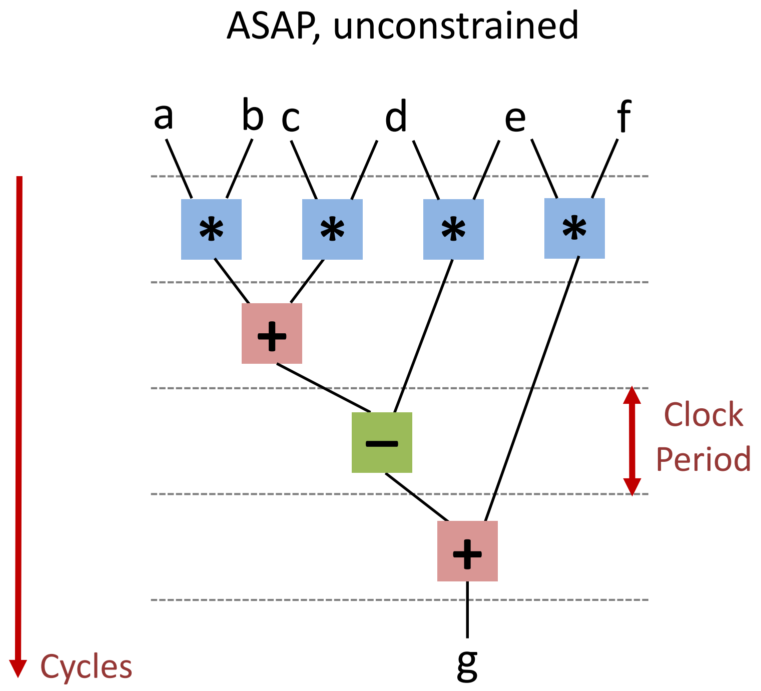
# Architectural Template

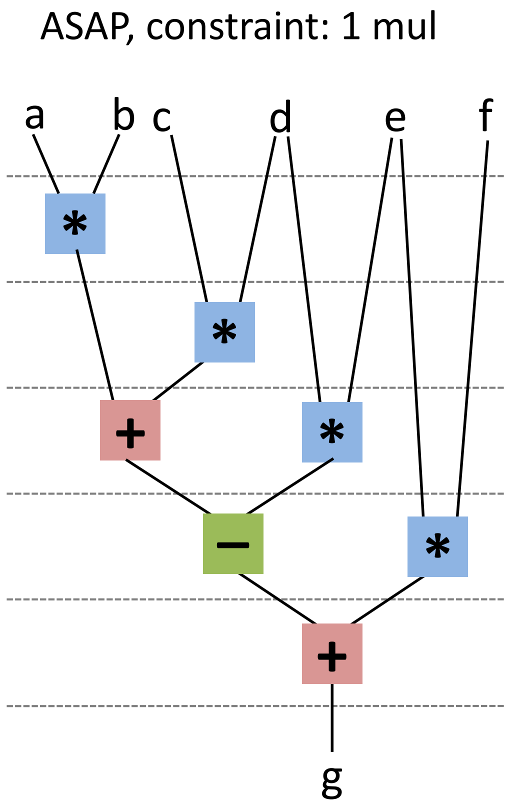
- We need to chose a template which we customize to and optimize for the code at hand
- Usually something of this sort:



# Scheduling the Datapath

- Assign operations to functional units respecting data dependencies and functional unit latencies





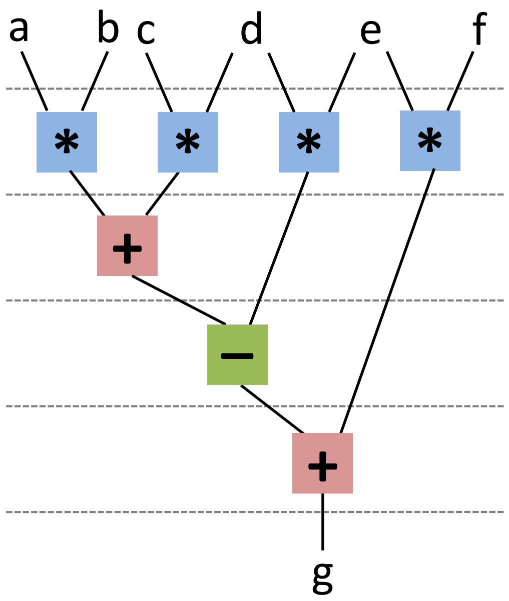
# Same as VLIW Scheduling?

- Very similar problem but with some notable differences:
  - Exact resources are not fixed; maybe there is a constraint on their total cost (e.g., area)
  - Clock cycle may be constrained but is in general not fixed; pipelining is not fixed (e.g., combinational operations can be chained)
  - No register file (which allows connecting everything to everything) but ad-hoc connectivity (variable cost and variable time impact)

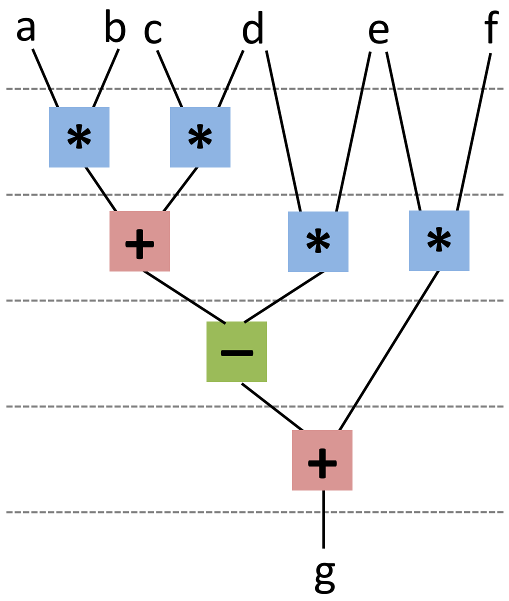
# Area Optimizations

- There may be cheaper ways to achieve the best latency
- New problem without immediate analogy in VLIWs

ASAP, unconstrained



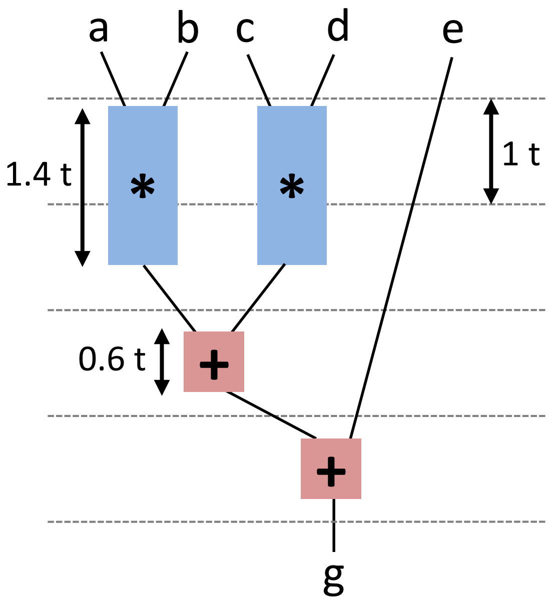
ASAP, constraint: 2 muls



# Chaining and Pipelining

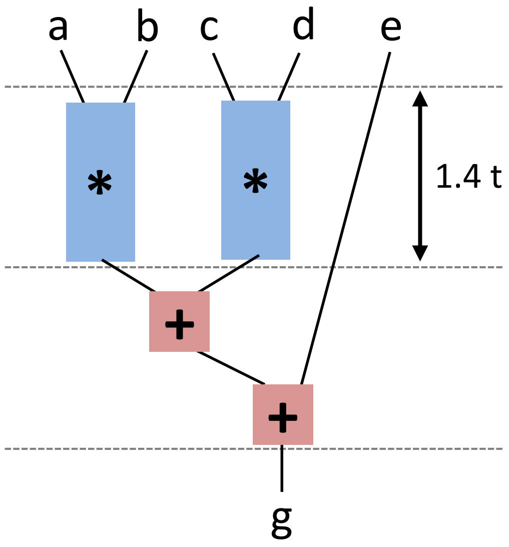
- Combinational operators can be chained and clock period can often be adjusted (shortest not necessarily fastest)
- Also, a new problem without immediate analogy in VLIWs

Before operation chaining and with fast clock



Total time: 4 × 1 t = **4 t**

After operation chaining and with slower clock



Total time: 2 × 1.4 t = **2.8 t**

# Scheduling under Resource Constraints

- Main focus of research in the early days
- The state of the art is based on the paper by Cong & Zhang, DAC 2006:
  - Given
    - A CDFG (i.e., a program)
    - A set of constraints including dependency constraints, resource constraints, latency constraints, cycle-time constraints, and relative timing constraints
  - Construct a valid schedule with minimal latency

- Used in recent tools such as Xilinx Vivado HLS
- But… is this all we need?

# Example: FIR

```
acc = 0;
for (i = 3; i >= 0; i--) {
   if (i == 0) {
      shift_reg[0] = x;
      acc += x * c[0];
   } else {
      shift_reg[i] = shift_reg[i-1];
      acc += shift_reg[i] * c[i];
   }
}
y = acc;
```

$x_k$

$x_k \ldots x_{k-3}$

$c_i$

$y_k$

$$y_k = \sum_{i=0}^{3} c_i x_{k-i}$$

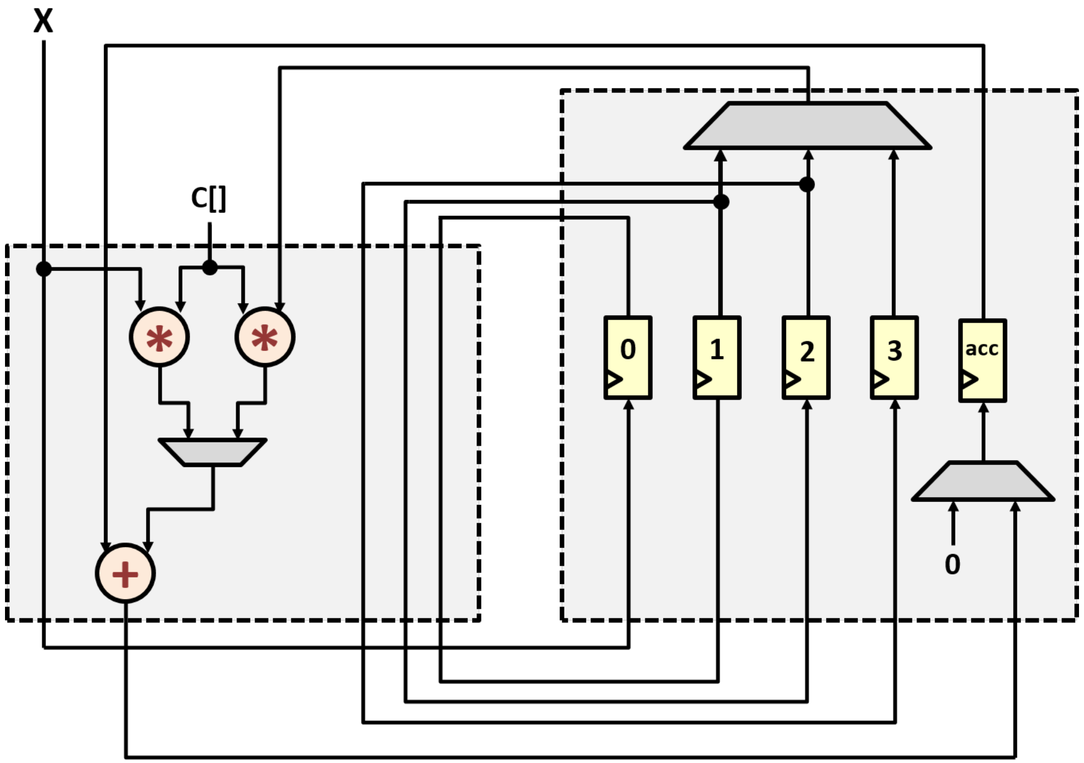
- The array **`shift_reg`** is static and represents the last 4 samples of **`x`**
- This could be in a function which receives a stream of **`x`** (the input signal) and produces at each call an element of **`y`** (the output signal)

Example inspired from Kastner, Matai, and Neuendorffer

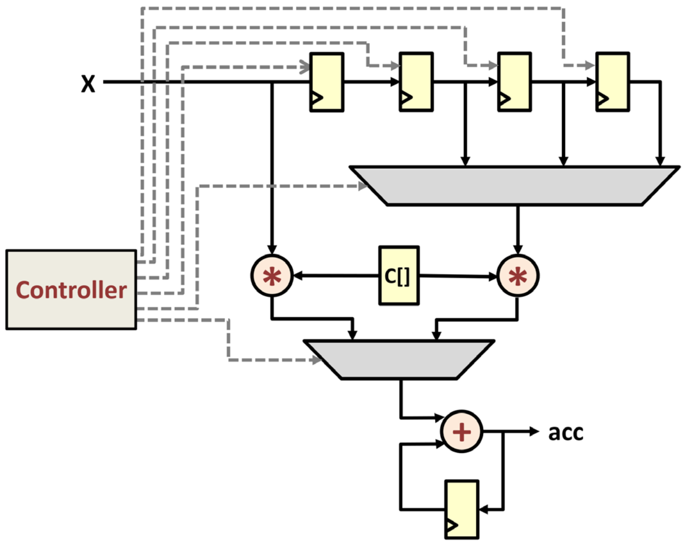
# A Literal Translation…

```
acc = 0;
for (i = 3; i >= 0; i--) {
   if (i == 0) {
      shift_reg[0] = x;
      acc += x * c[0];
   } else {
      shift_reg[i] = shift_reg[i-1];
      acc += shift_reg[i] * c[i];
   }
}
y = acc;
```

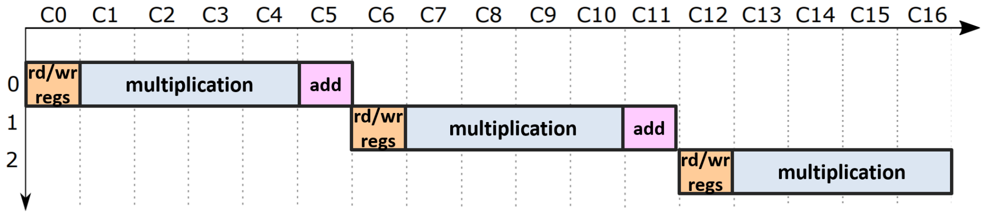
1. If-convert control flow whenever possible
2. Implement all existing registers
3. Implement datapath for all BBs
4. Create steering wires and muxes to connect everything



# Naïve FIR



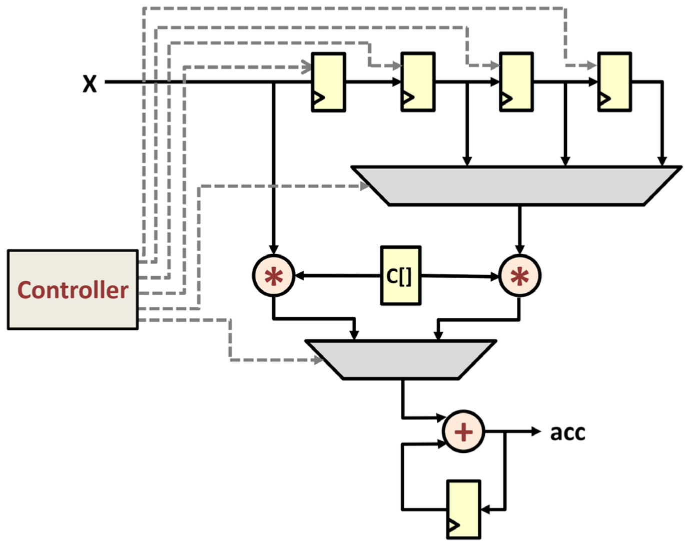
```
acc = 0;
for (i = 3; i >= 0; i--) {
   if (i == 0) {
      shift_reg[0] = x;
      acc += x * c[0];
   } else {
      shift_reg[i] = shift_reg[i-1];
      acc += shift_reg[i] * c[i];
   }
}
y = acc;
```



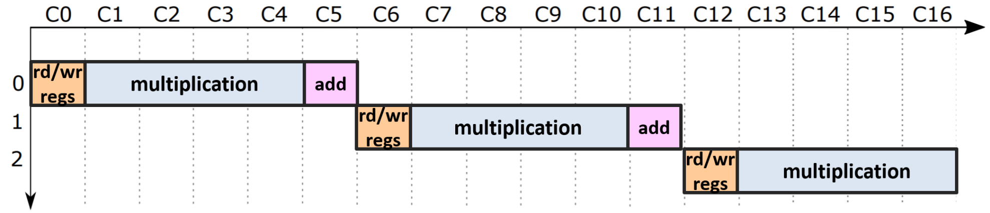
# Manual Code Refactoring

- Direct results are very often highly suboptimal
  - See FIR example
- Users should have a sense of what circuit they want to produce and suggest it to HLS tools by restructuring the code
  - See coming slides
- HLS tools today are **not** really meant to **abstract away hardware design issues** from software programmers; in practice, they are more like productivity tools to help hardware designers explore quickly the space of hardware designs they may wish to produce

# Naïve FIR

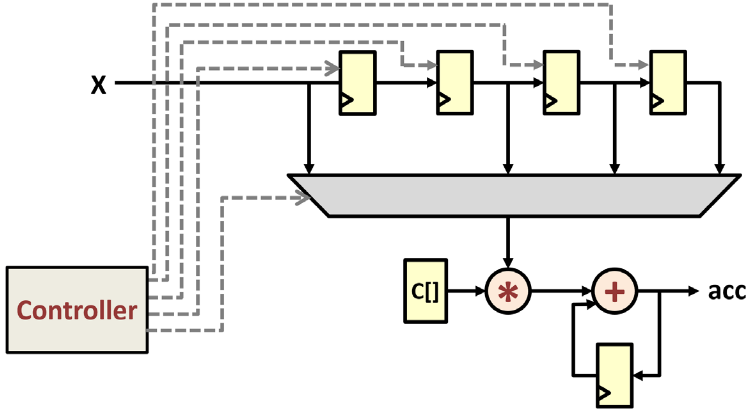


```
acc = 0;
for (i = 3; i >= 0; i--) {
   if (i == 0) {
      shift_reg[0] = x;
      acc += x * c[0];
   } else {
      shift_reg[i] = shift_reg[i-1];
      acc += shift_reg[i] * c[i];
   }
}
y = acc;
```

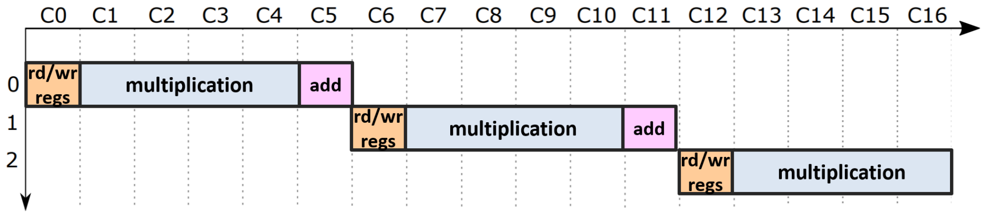


- We are always computing both sides of the control decision, but which one is needed in a particular iteration is perfectly evident

# Loop Peeling

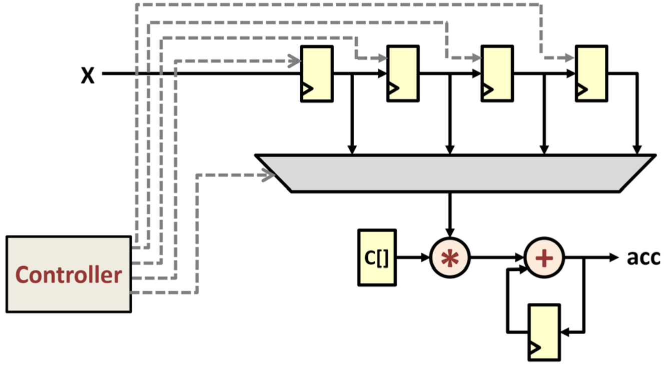


```
acc = 0;

for (i = 3; i > 0; i--) {
   shift_reg[i] = shift_reg[i-1];
   acc += shift_reg[i] * c[i];
}

shift_reg[0] = x;
acc += x * c[0];

y = acc;
```

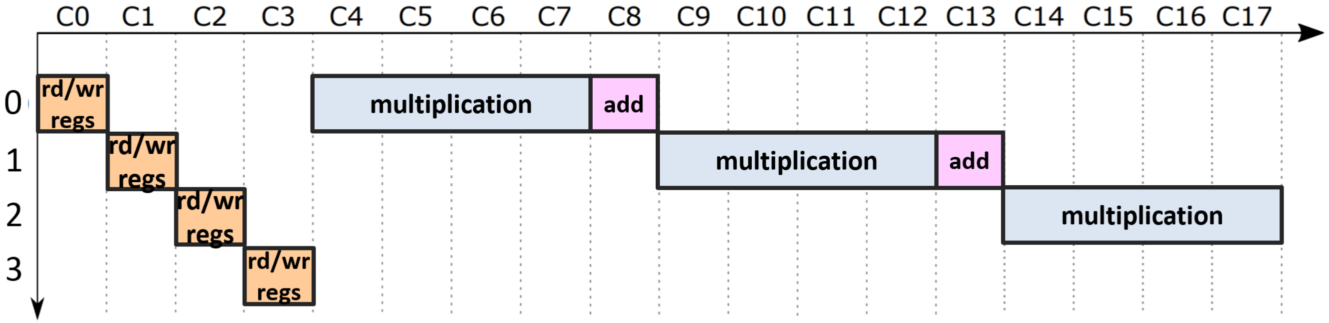


- The loop is doing two tasks completely independent from each other (shifting the signal samples and computing the new output sample), so shall we split it into two loops?

# Loop Fission



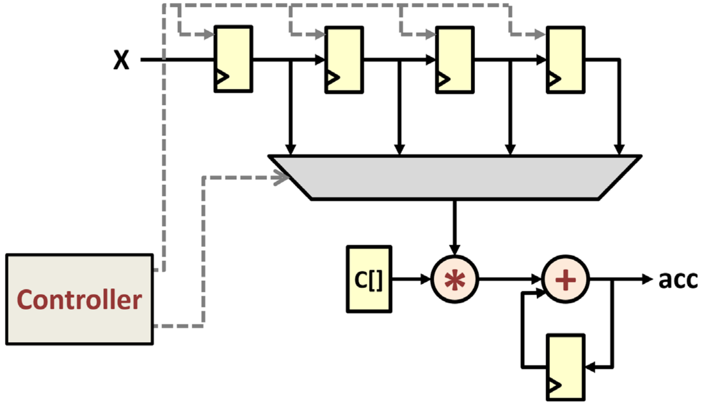
```
for (i = 3; i > 0; i--) {
   shift_reg[i] = shift_reg[i-1];
}
shift_reg[0] = x;

acc = 0;
for (i = 3; i >= 0; i--) {
   acc += shift_reg[i] * c[i];
}
y = acc;
```

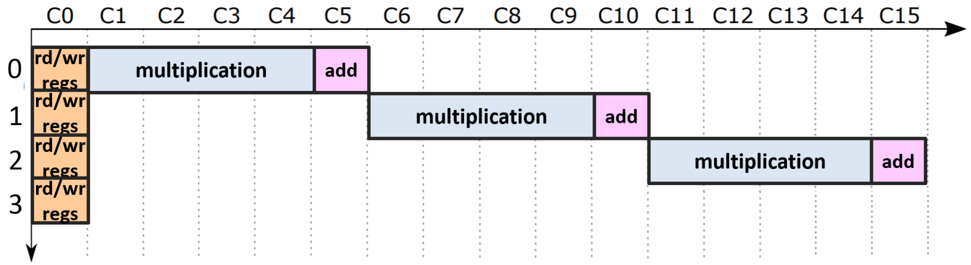


- Not terribly useful per se, just two independent and parallel machines
- Does this create an opportunity to unroll loop 1? Note that it contains no computation…

# Loop Unrolling (loop 1)

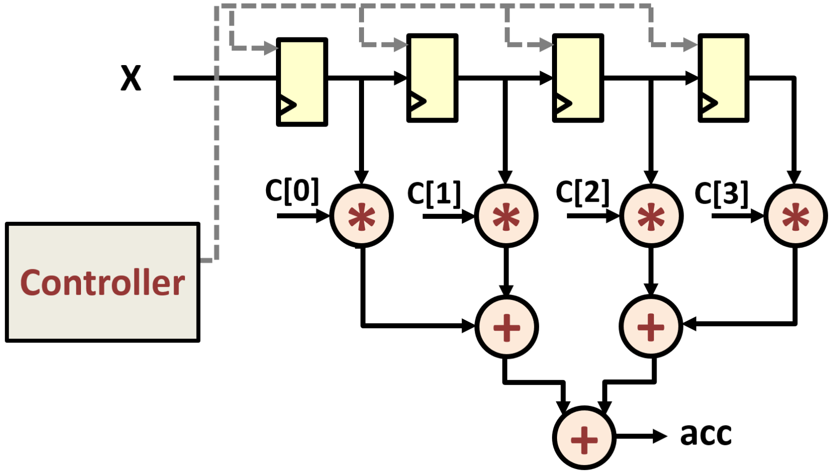
```
shift_reg[3] = shift_reg[2];
shift_reg[2] = shift_reg[1];
shift_reg[1] = shift_reg[0];
shift_reg[0] = x;

acc = 0;
for (i = 3; i >= 0; i--) {
    acc += shift_reg[i] * c[i];
}
y = acc;
```



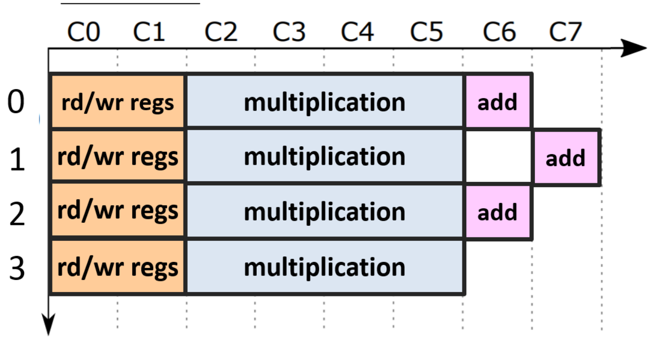


- Loop 1 has become a “pipeline” (although a fairly degenerate one) by unrolling—this is certainly desirable regardless
- Loop 2 is not pipelined: the initiation interval is exactly equal to the latency of the kernel—unroll?

# Loop Unrolling (loop 2)



```
shift_reg[3] = shift_reg[2];
shift_reg[2] = shift_reg[1];
shift_reg[1] = shift_reg[0];
shift_reg[0] = x;

acc = shift_reg[3] * c[3];
acc += shift_reg[2] * c[2];
acc += shift_reg[1] * c[1];
acc += shift_reg[0] * c[0];

y = acc;
```



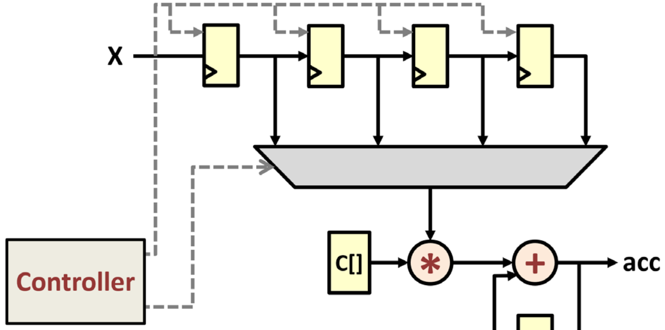
- De facto, a new iteration now starts every cycle
- But resources may be too much—and partial unrolling would achieve some pipelining but yet it would still fill and drain the pipeline every iteration

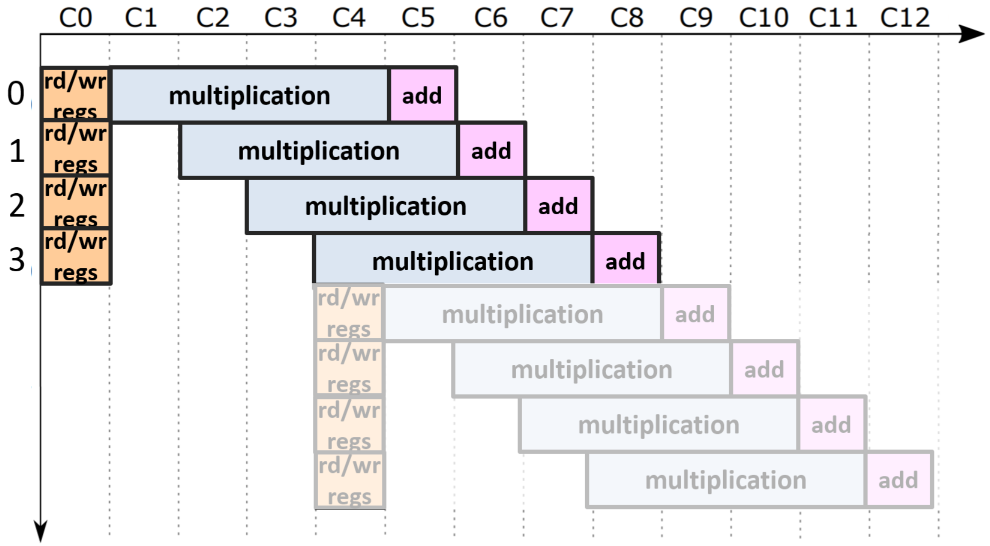
# Pipelining

- Perfect pipelining cannot be achieved easily by rewriting the code
- We need to schedule differently the operations within a loop so that operations of different iterations take place simultaneously
- Remember “software pipelining”? Now we need it so that a software program represents a hardware pipeline
- HLS needs to implement some form of modulo scheduling

# Pipelining Result

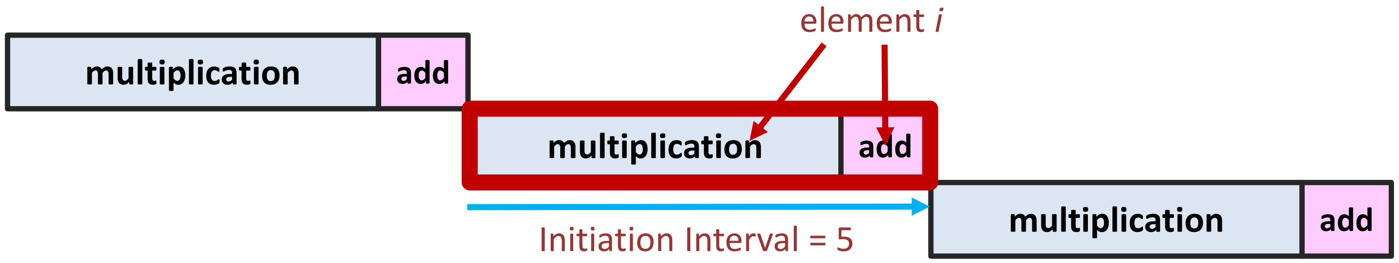
```
shift_reg[3] = shift_reg[2];
shift_reg[2] = shift_reg[1];
shift_reg[1] = shift_reg[0];
shift_reg[0] = x;

acc = 0;
for (i = 3; i >= 0; i--) {
    #pragma HLS pipeline
    acc += shift_reg[i] * c[i];
}
y = acc;
```

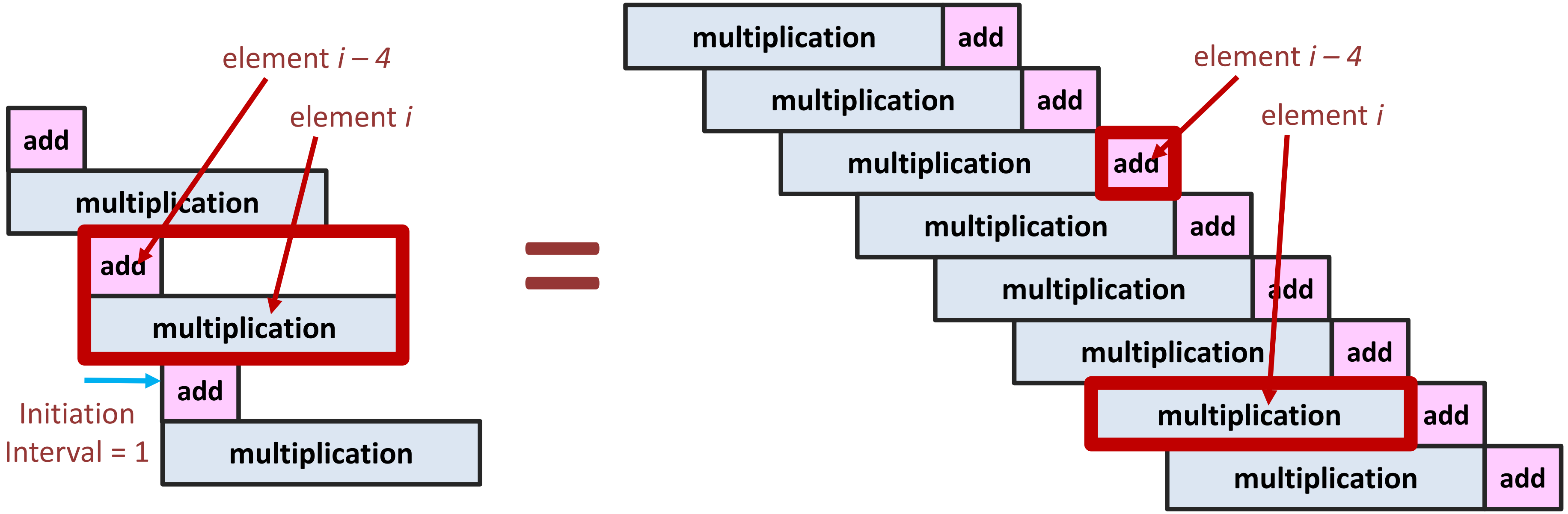




- One output sample produced every 4 cycles and minimal resources

# Loop Restructuring as with VLIWs





# Classic HLS and VLIW Compilation

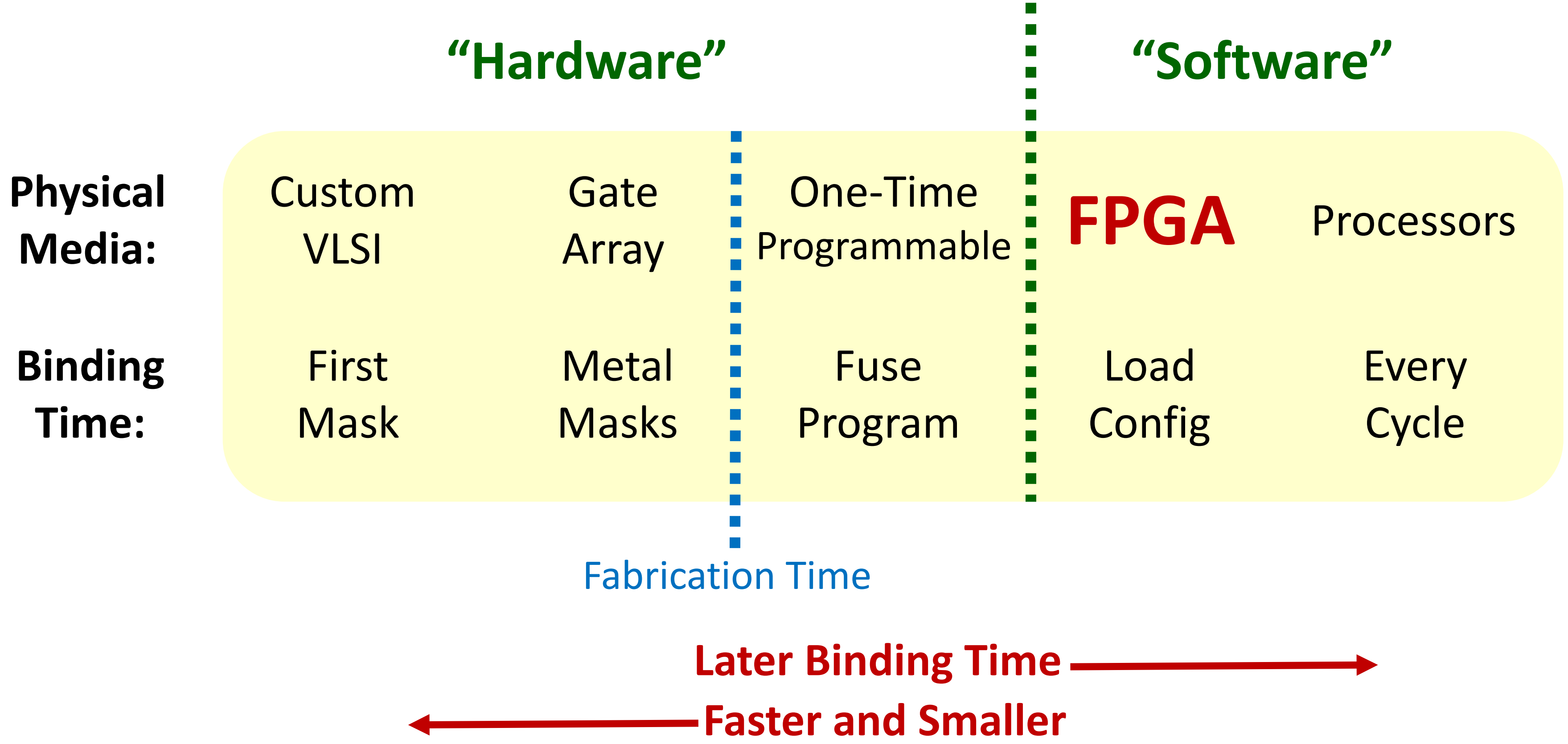
- Striking resemblance of the two undertakings
  - Both try to produce a **static schedule** of operations
  - Both try to reduce to a minimum **control decisions**
- Both suffer from **similar limitations**: they cope poorly with variability including variable latency operations, uncertain events—such as memory dependencies, unpredictable control flow (see part 3)
- Both impose **burdens onto the user**: decisions on how and where to apply optimizations are not self-evident, depend on the particular combination of user constraints (note that the solution space is much wider for HLS), and thus are often left to users through code restructuring or pragmas (see HLS lab)

# Extent of Programming Language Support

- Complete support for C/C++? Not quite:
  - No dynamic memory allocation (no malloc(), etc.)
    - Research work on providing such primitives for FPGA accelerators in high-end systems, for instance
  - No recursion
  - Limited use of pointers-to-pointers
  - No system calls (no printf(), etc.)
  - Other limitations related to the ability to determine critical details (e.g., function interfaces) at compile time
- Details vary from HLS tool to HLS tool
  - Perhaps similarly to the early days of logic synthesis (which part of VHDL is supported and with what exact meaning?)

# Where Has *Programmability* Gone?

- **FPGAs** are an (increasingly?) important “programmable” technology in the hardware ↔ software spectrum
- **Early binding** time gives performance and/or cost advantages



| | “Hardware” | | | “Software” | |
|---|---|---|---|---|---|
| **Physical Media:** | Custom VLSI | Gate Array | One-Time Programmable | **FPGA** | Processors |
| **Binding Time:** | First Mask | Metal Masks | Fuse Program | Load Config | Every Cycle |

Fabrication Time

**Later Binding Time** →

← **Faster and Smaller**

Adapted from DeHon, © ACM 1999

# 3

## Dynamically Scheduled High-Level Synthesis

*(with Lana Josipović)*

# High-level Synthesis and Static Scheduling

- **High-level synthesis (HLS)** may be the future of reconfigurable computing
  - Design circuits from high-level programming languages

- As seen in Part 2, classic HLS relies on **static schedules**
  - Each operation executes at a cycle fixed at synthesis time

- Scheduling dictated by compile-time information
  - Maximum parallelism in regular applications
  - Limited parallelism when information unavailable at compile time (i.e., latency, memory or control dependencies)

# Part 3 Outline

## What traditional HLS does not do well

Synthesis of dataflow circuits

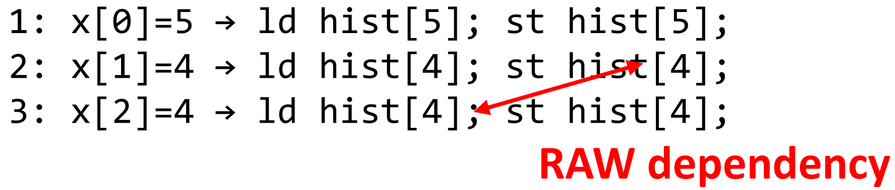
Buffers and performance

The problem with memory

Conquering new grounds with speculation

# The Limitations of Static Scheduling

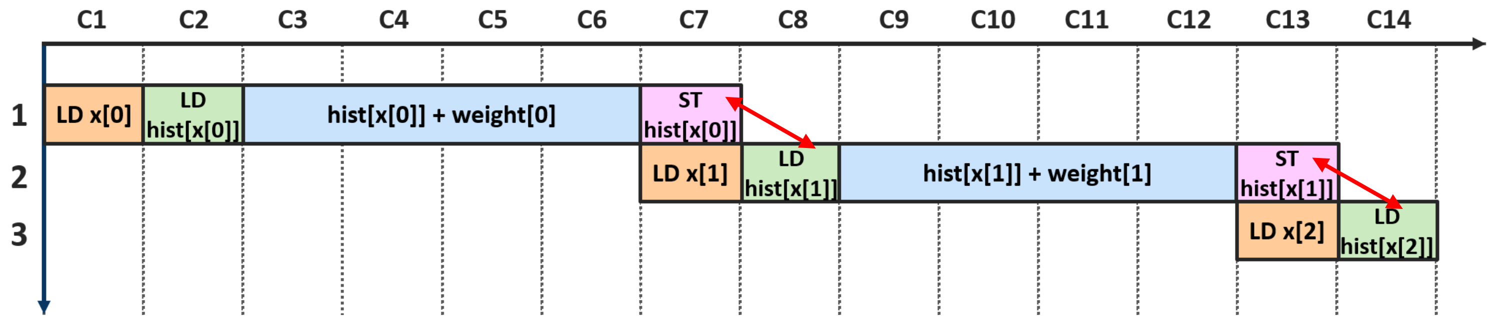
```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```



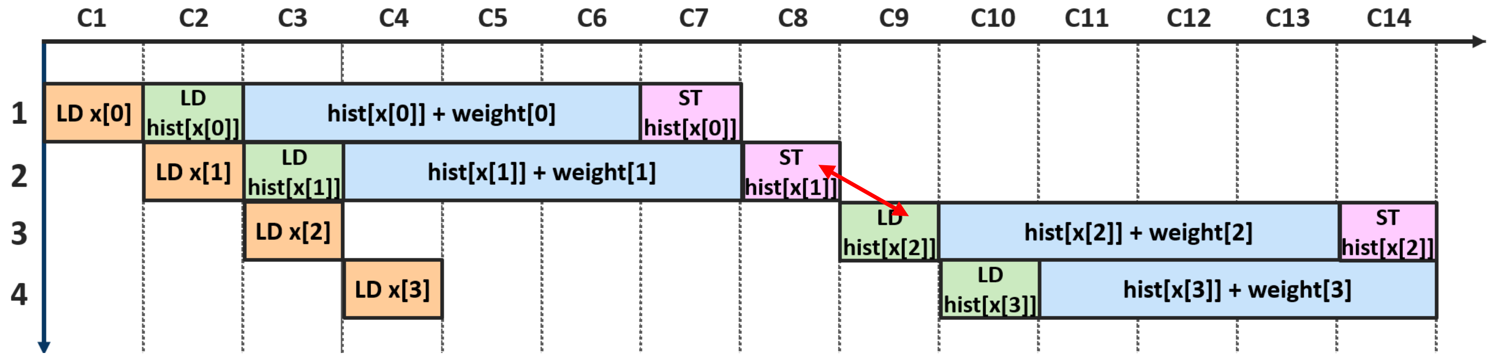
```
1: x[0]=5 → ld hist[5]; st hist[5];
2: x[1]=4 → ld hist[4]; st hist[4];
3: x[2]=4 → ld hist[4]; st hist[4];
```

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



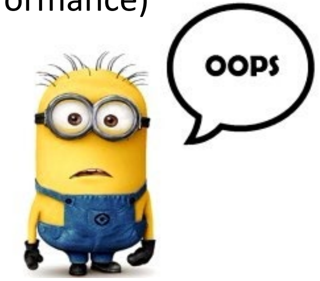
# Statically vs. Dynamically Scheduled

| | **Statically Scheduled**<br>→ "Compiler does the job" | **Dynamically Scheduled**<br>→ "Hardware does the job" |
|---|---|---|
| Computer Architecture | **VLIW Processors** | **Out-of-Order Superscalar Processors** |

**Great for some embedded applications**
(expert developers, heavy manual code refactoring and optimizations, etc.)

**Catastrophic for general purpose**
(out-of-the-box compilation fails to deliver high performance)

intel
Itanium
inside



# Statically vs. Dynamically Scheduled

| | **Statically Scheduled**<br>→ “Compiler does the job” | **Dynamically Scheduled**<br>→ “Hardware does the job” |
|---|---|---|
| Computer Architecture | **VLIW Processors** | **Out-of-Order Superscalar Processors** |
| High-Level Synthesis | **Traditional HLS** | **???** |

# Part 3 Outline

✓ What traditional HLS does not do well

## Synthesis of dataflow circuits

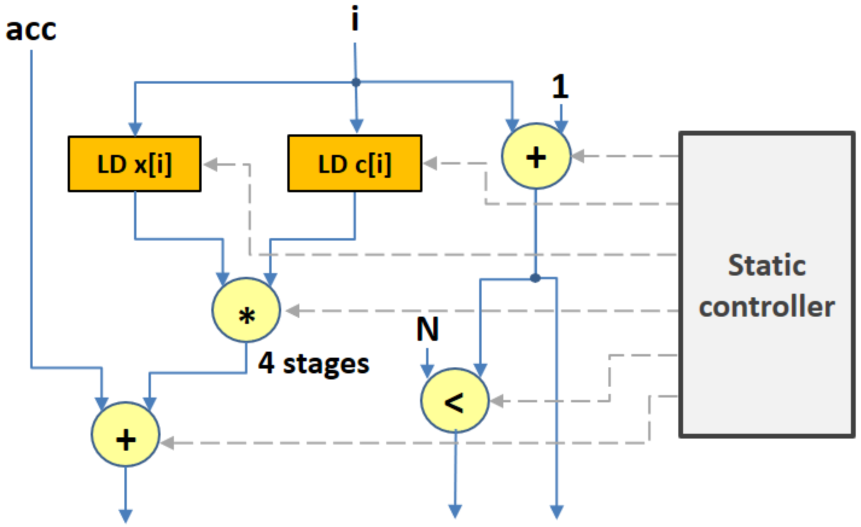
Buffers and performance

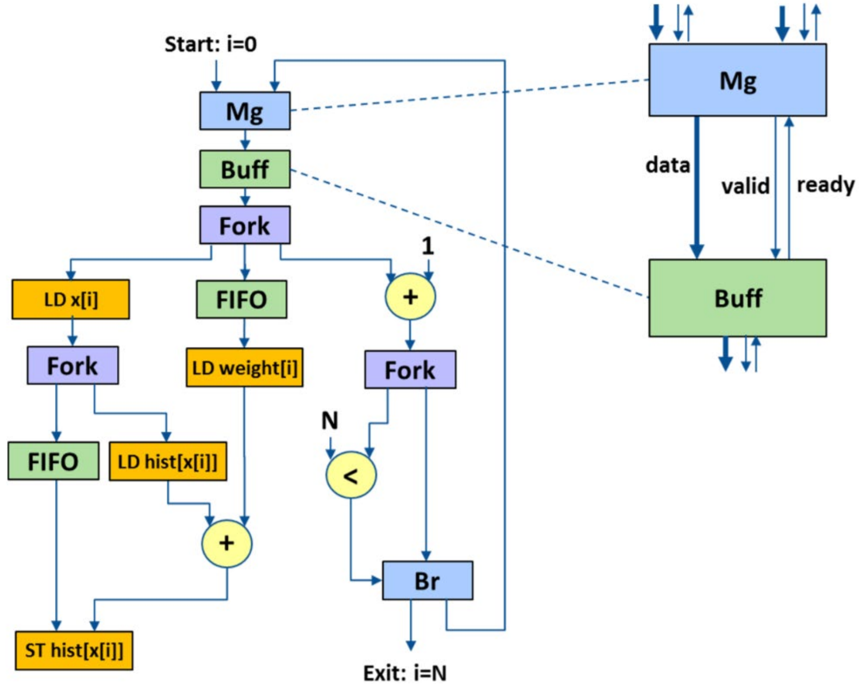
The problem with memory

Conquering new grounds with speculation

# A Different Way to Do HLS

- Refrain from triggering the operations through a centralized pre-planned controller
- **Make scheduling decisions at runtime**: as soon as all conditions for execution are satisfied, an operation starts





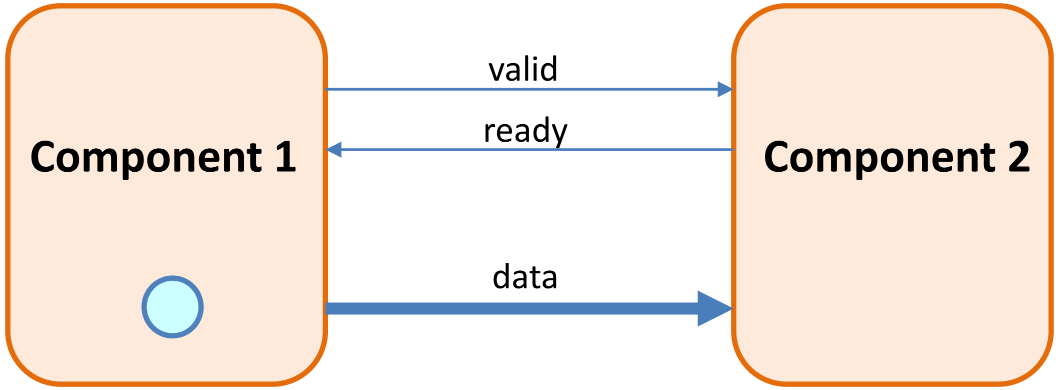
# A Different Way to Do HLS

- **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 et al. Bounded dataflow networks and latency-insensitive circuits. MEMOCODE'09.

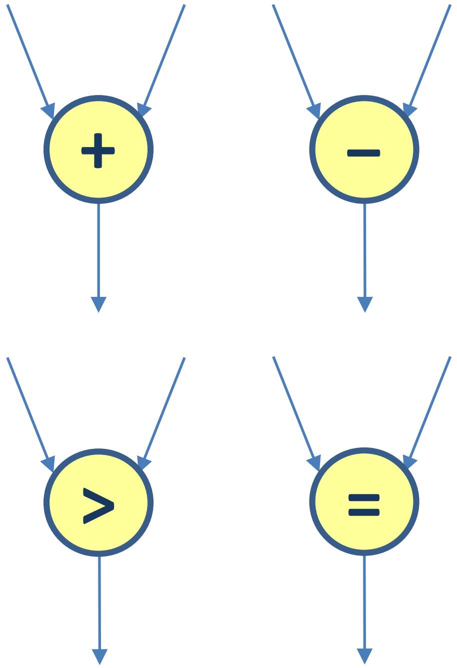
**How to create dataflow circuits from high-level programs?**

# Dataflow Circuits

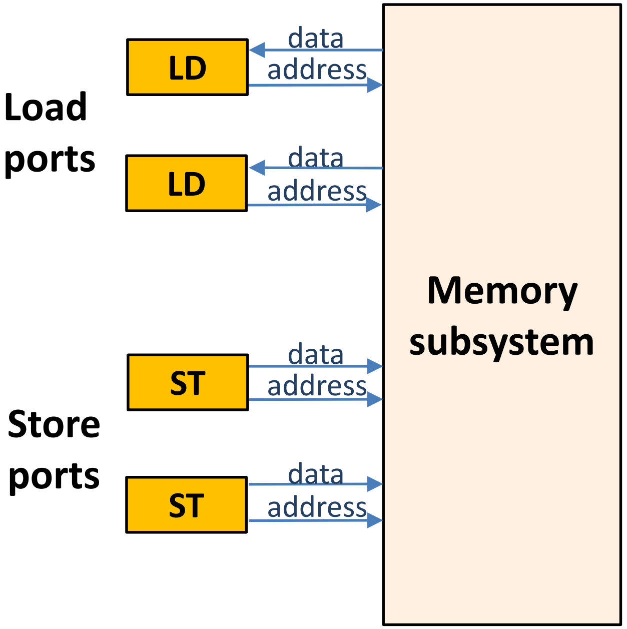
- Example using the **SELF (Synchronous ELastic Flow)** protocol
  - Cortadella et al. Synthesis of synchronous elastic architectures. DAC’06.
- Every component communicates via a pair of handshake signals
- The data is propagated from component to component as soon as memory and control dependencies are resolved



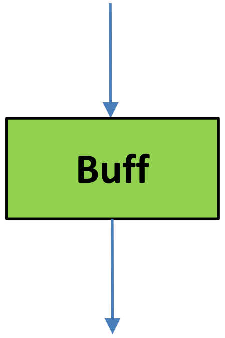
# Dataflow Components



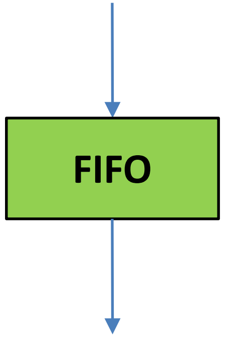
Functional units



Memory interface

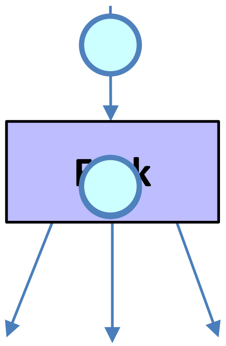


Buffer

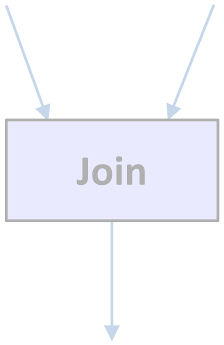


FIFO

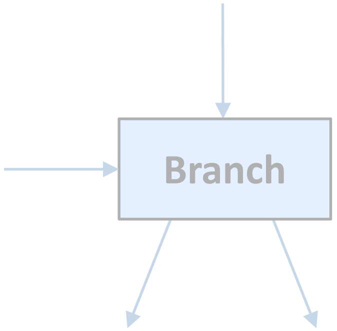
# Dataflow Components



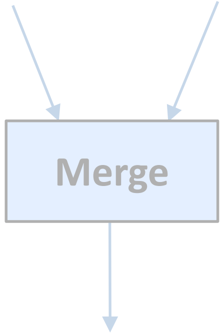
Fork



Join

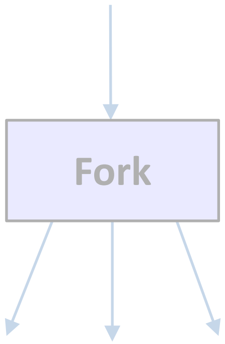


Branch

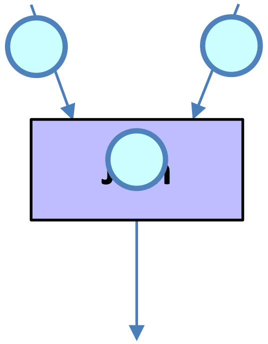


Merge

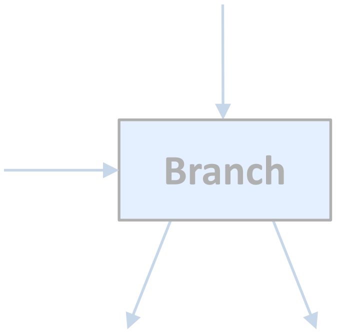
# Dataflow Components



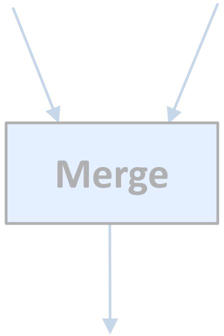
Fork



**Join**

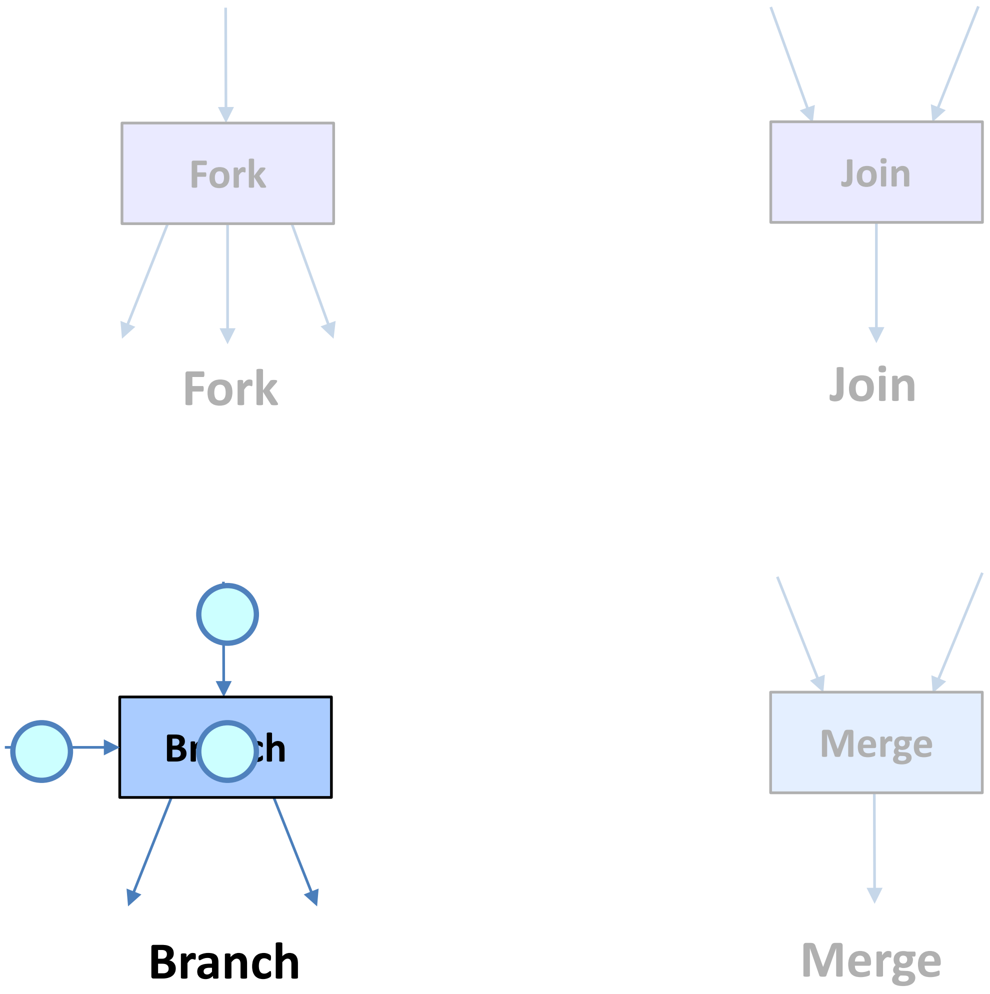


Branch

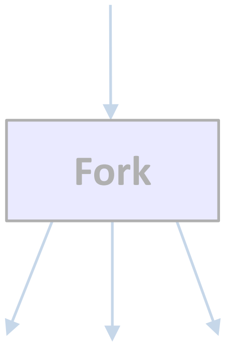


Merge

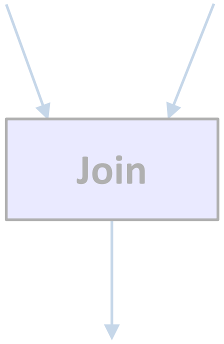
# Dataflow Components



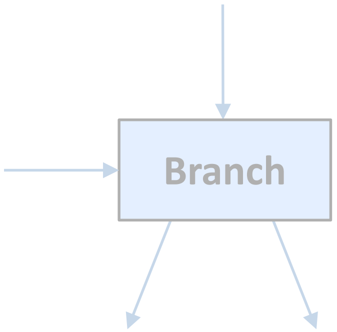
# Dataflow Components



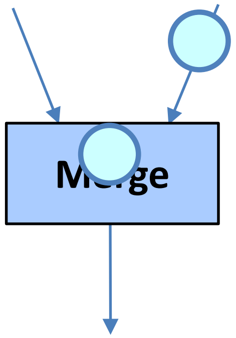
Fork



Join



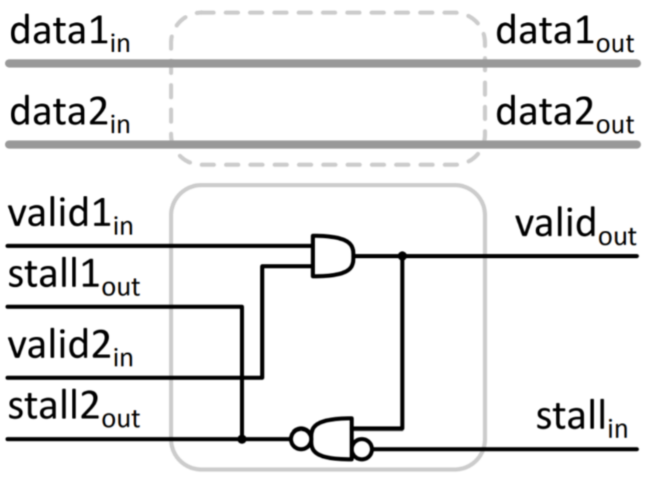
Branch



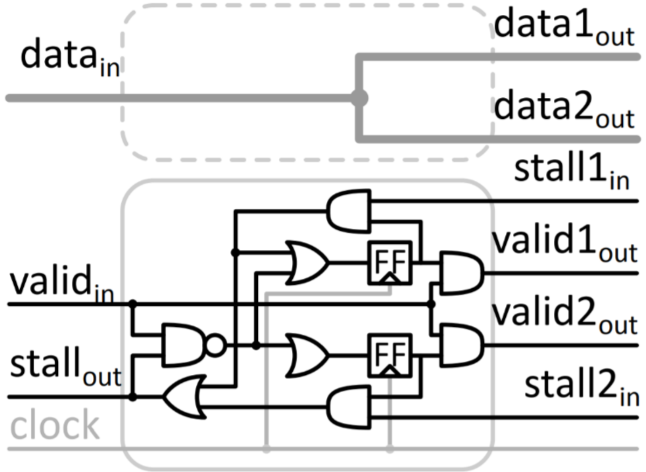
Merge

# Dataflow Components

- Although inspired by asynchronous circuits, elastic circuits are strictly synchronous and perfectly adapted to traditional VLSI and FPGA flows



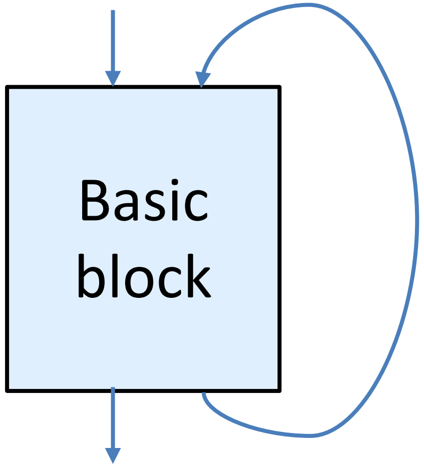
**Join**



**Eager Fork**

# Synthesizing Dataflow Circuits

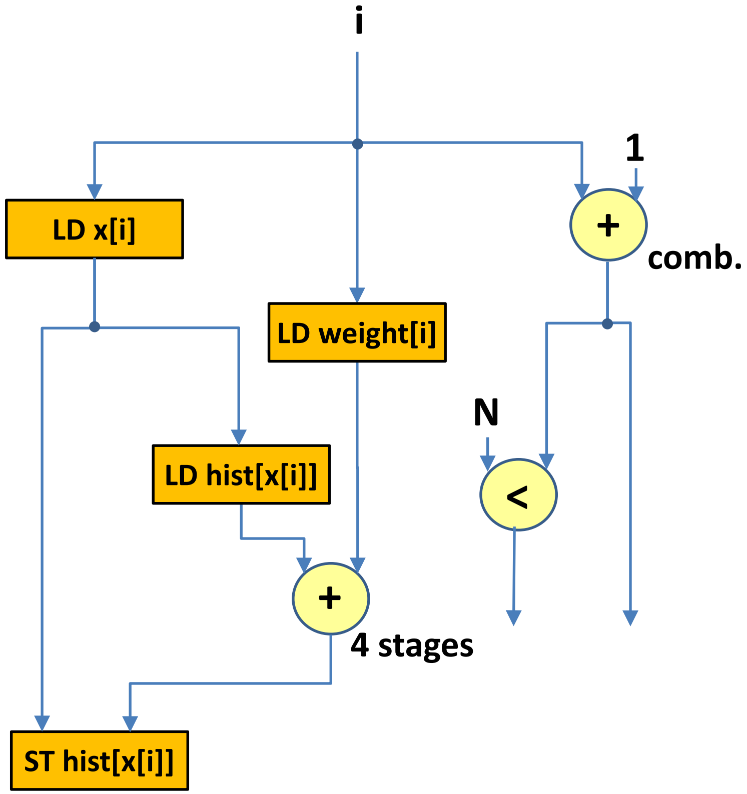
- C to intermediate graph representation
  - LLVM compiler framework



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

# Synthesizing Dataflow Circuits

- Constructing the datapath

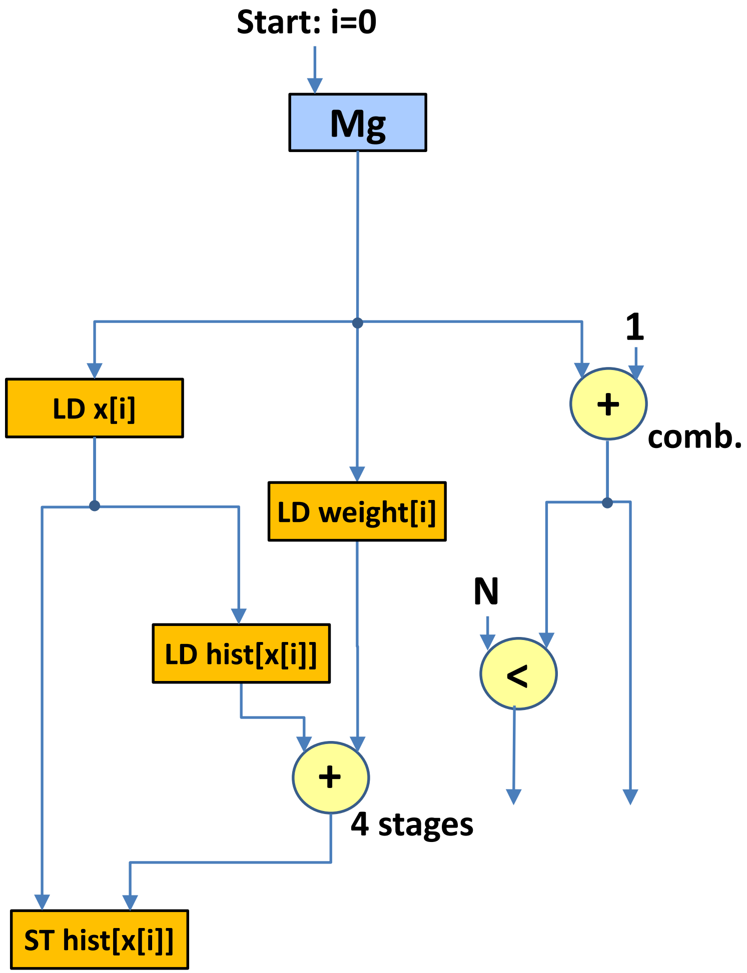


```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

**Each operator corresponds to a functional unit**

# Synthesizing Dataflow Circuits

- Implementing control flow

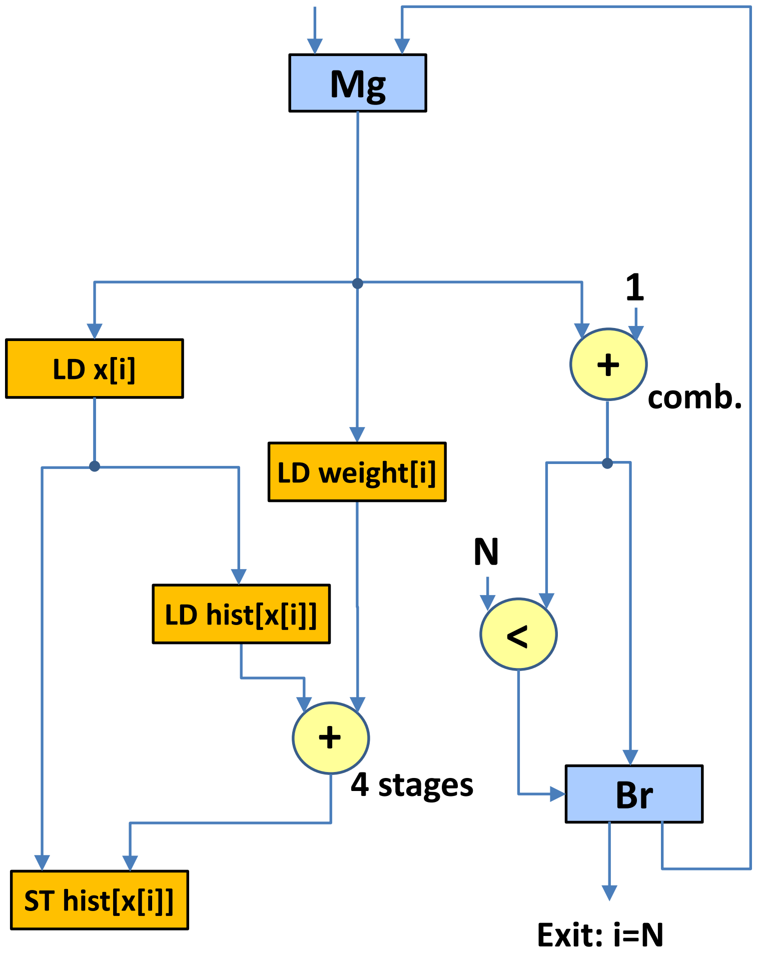


```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

A Merge for each variable entering the BB

# Synthesizing Dataflow Circuits

- Implementing control flow

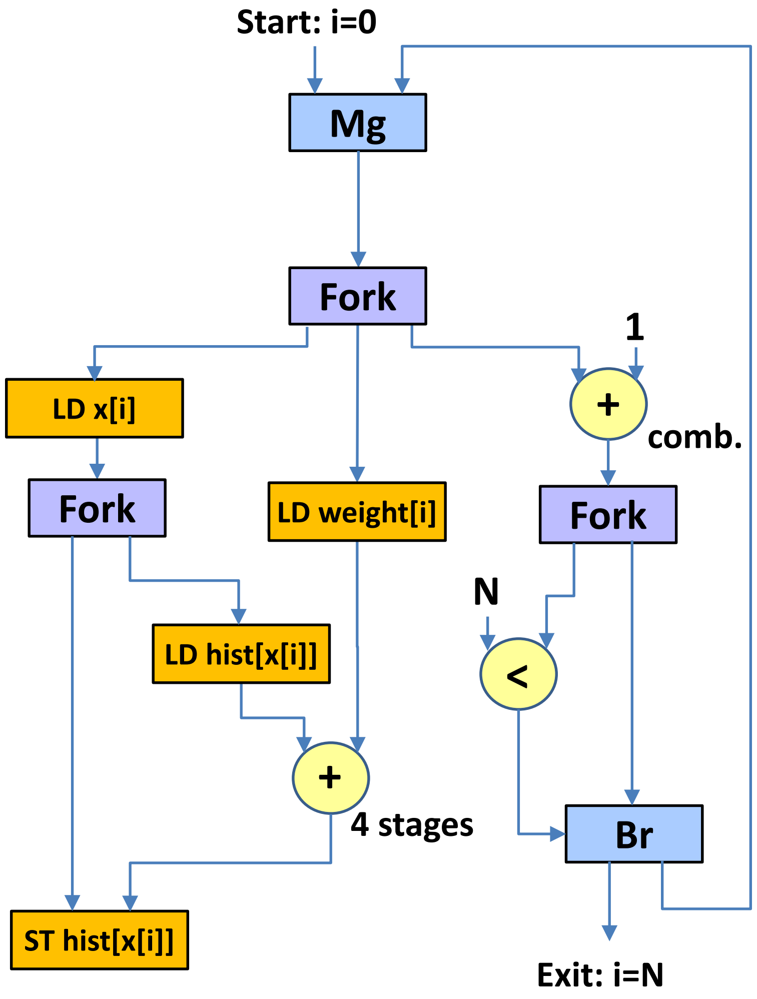


```
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
```

**A Branch for each variable exiting the BB**

# Synthesizing Dataflow Circuits

- Inserting Forks



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

A Fork after every node with multiple successors

# Part 3 Outline

✓ What traditional HLS does not do well

✓ Synthesis of dataflow circuits

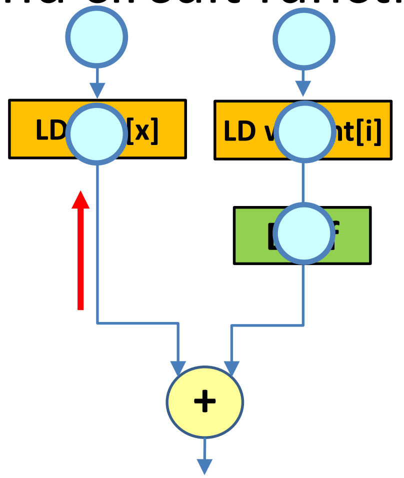
## Buffers and performance

The problem with memory

Conquering new grounds with speculation

# Adding Buffers

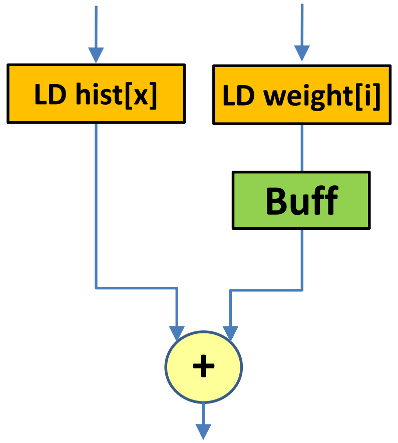
- Buffers and circuit functionality



**Buffer insertion does not affect circuit functionality**

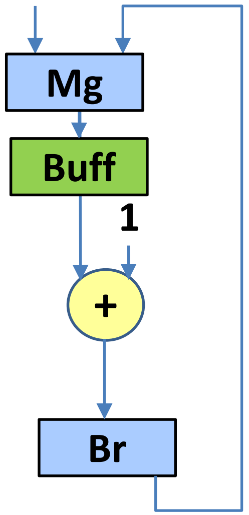
# Adding Buffers

- Buffers and circuit functionality



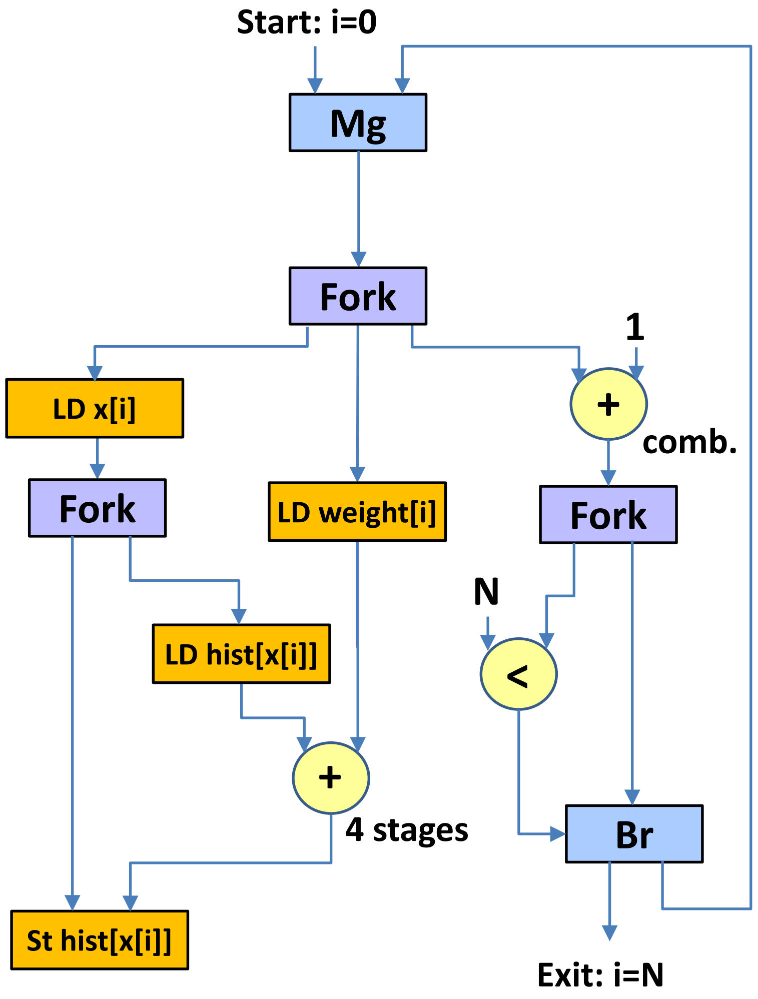
Buffer insertion does not affect circuit functionality

- Buffers and avoiding deadlock

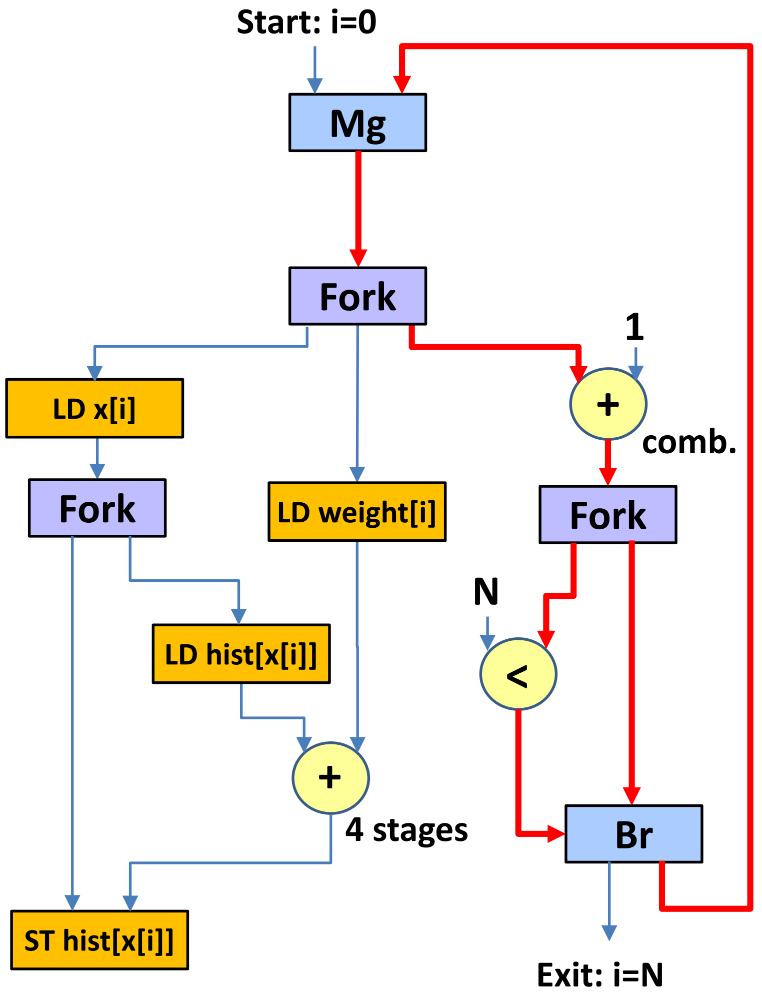


Each combinational loop in the circuit needs to contain at least one buffer

# Adding Buffers

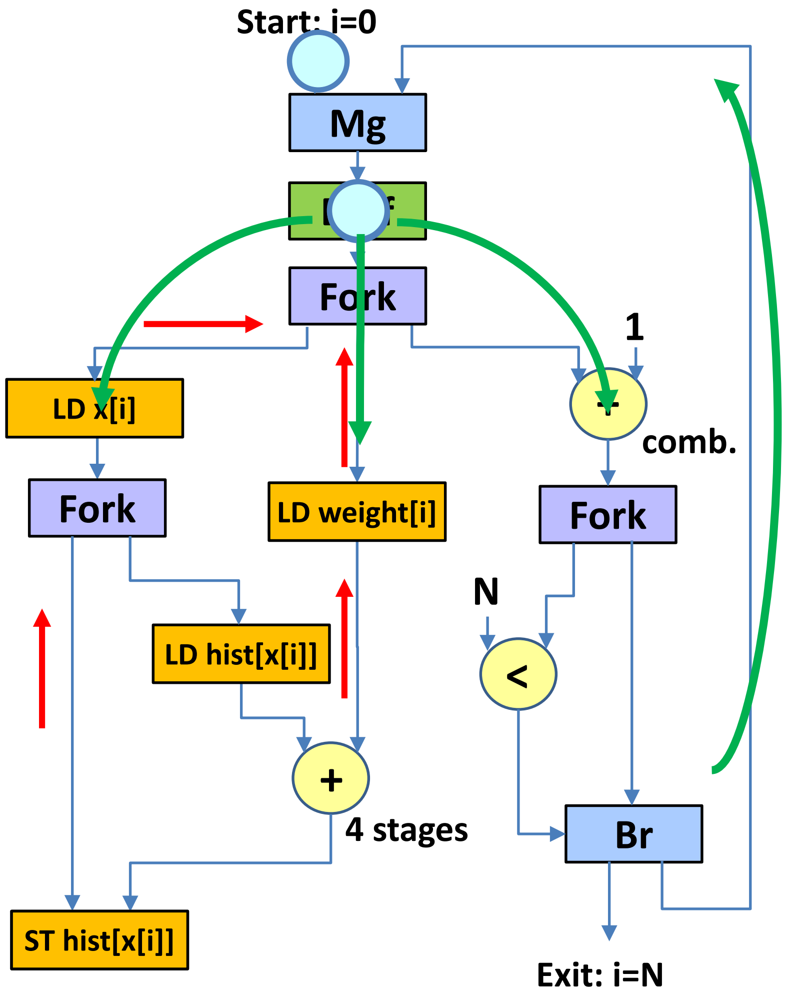


# Adding Buffers

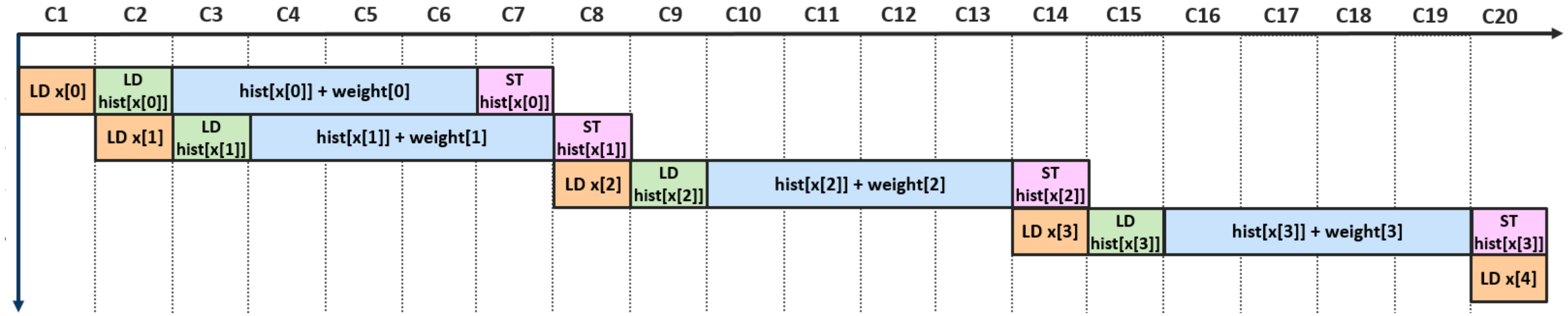


Two combinational loops

# Adding Buffers



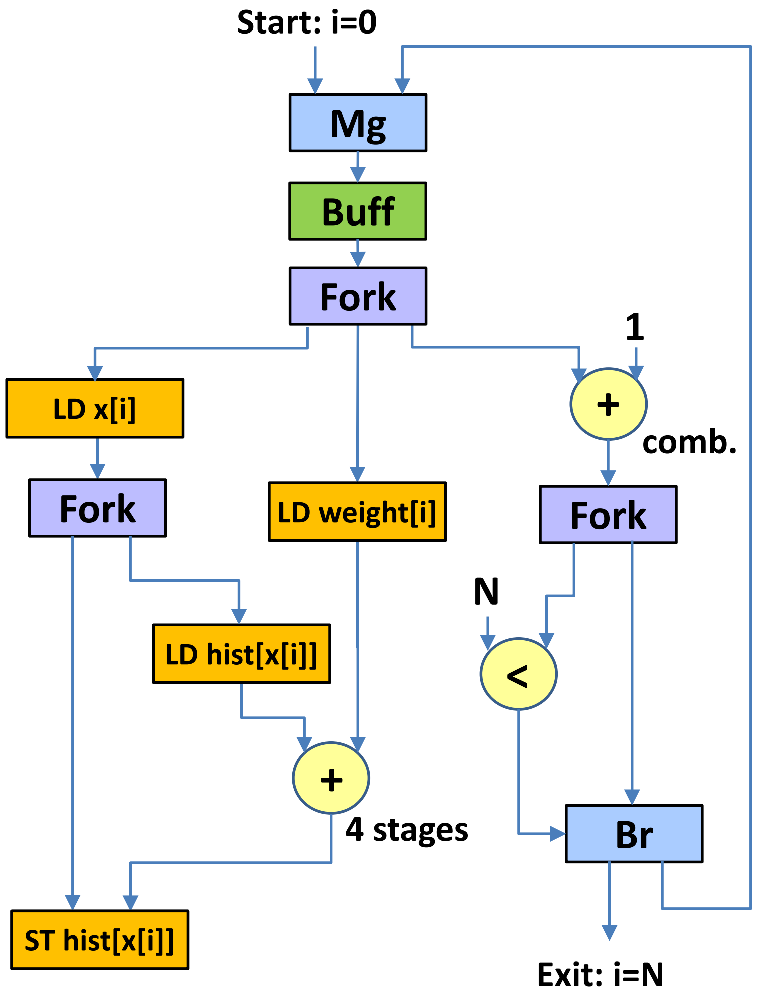
# Adding Buffers



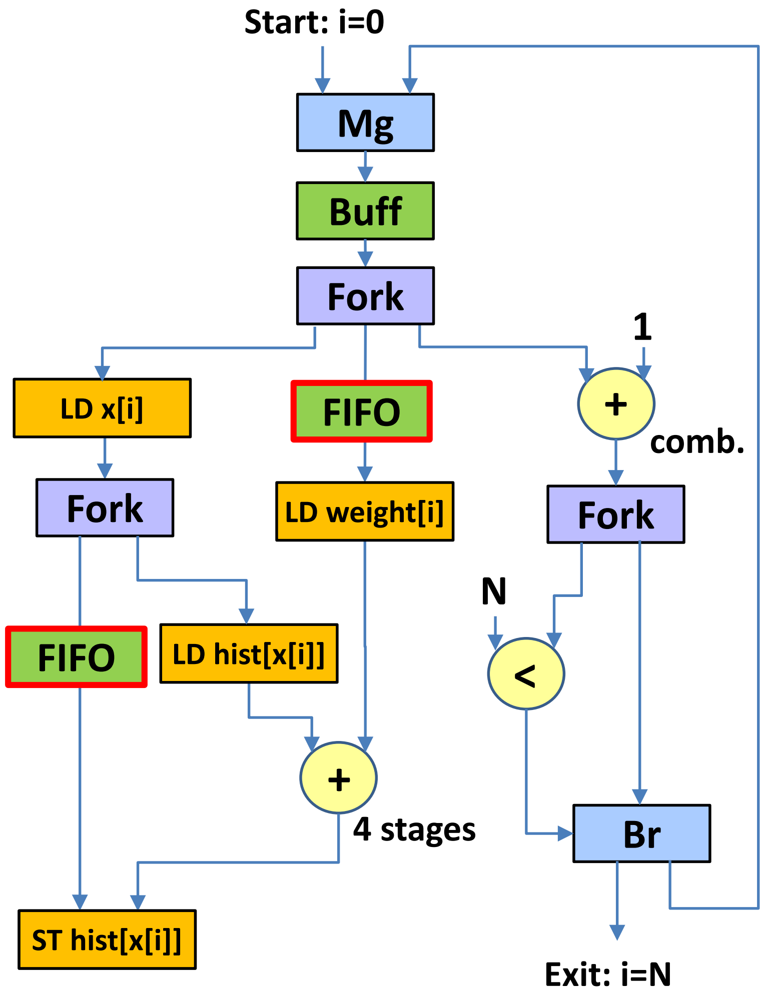
| C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | C11 | C12 | C13 | C14 | C15 | C16 | C17 | C18 | C19 | C20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LD x[0] | LD hist[x[0]] | hist[x[0]] + weight[0] | | | | ST hist[x[0]] | | | | | | | | | | | | | |
| | LD x[1] | LD hist[x[1]] | hist[x[1]] + weight[1] | | | | ST hist[x[1]] | | | | | | | | | | | | |
| | | | | | | | LD x[2] | LD hist[x[2]] | hist[x[2]] + weight[2] | | | | ST hist[x[2]] | | | | | | |
| | | | | | | | | | | | | | LD x[3] | LD hist[x[3]] | hist[x[3]] + weight[3] | | | | ST hist[x[3]] |
| | | | | | | | | | | | | | | | | | | | LD x[4] |

**Backpressure from slow paths prevents pipelining**

# Optimizing Performance

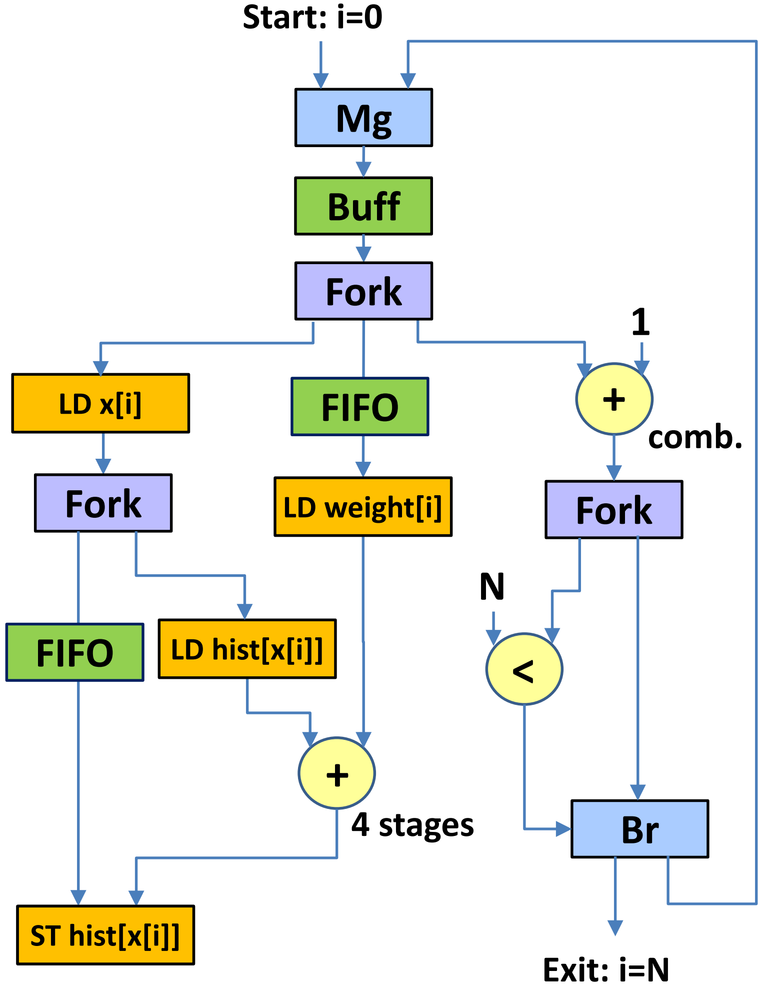


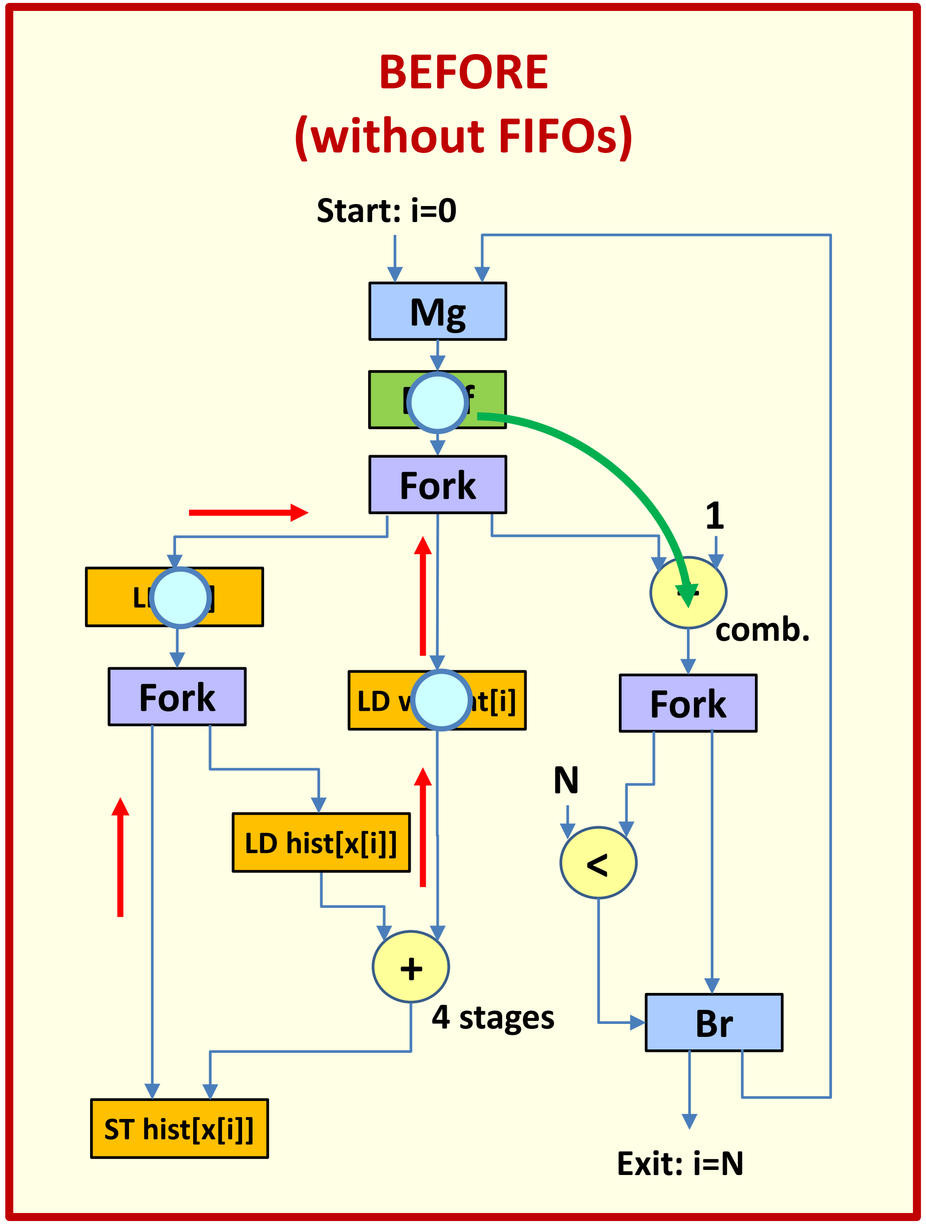
# Optimizing Performance



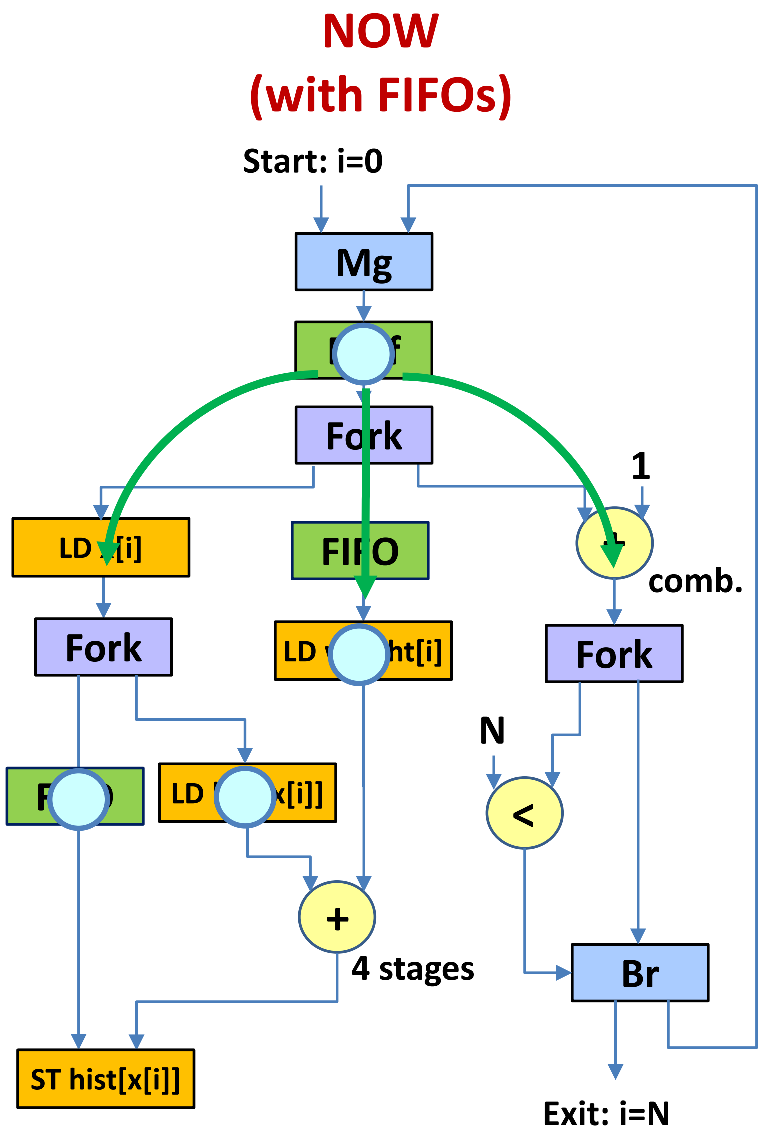
Insert FIFOs into slow paths

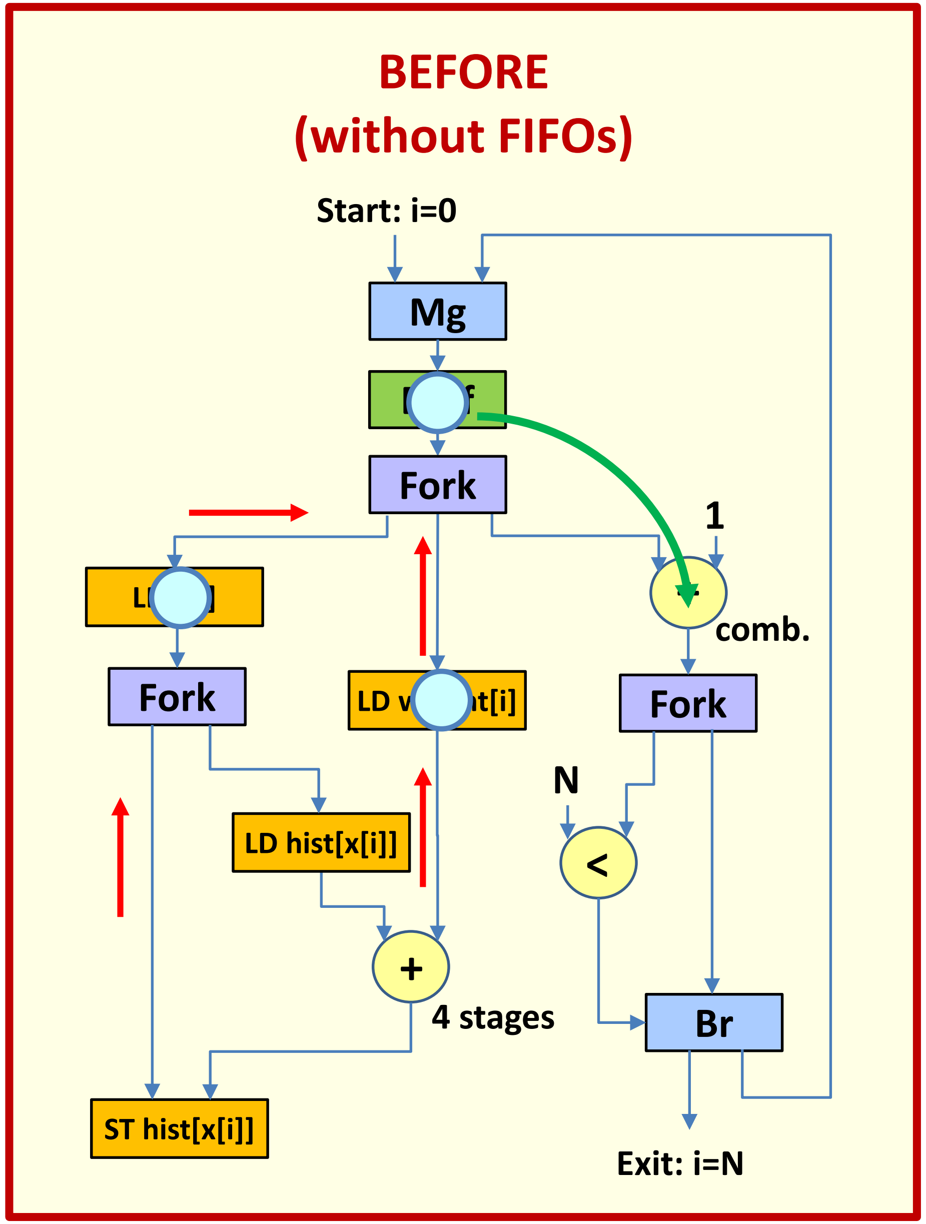
# Optimizing Performance



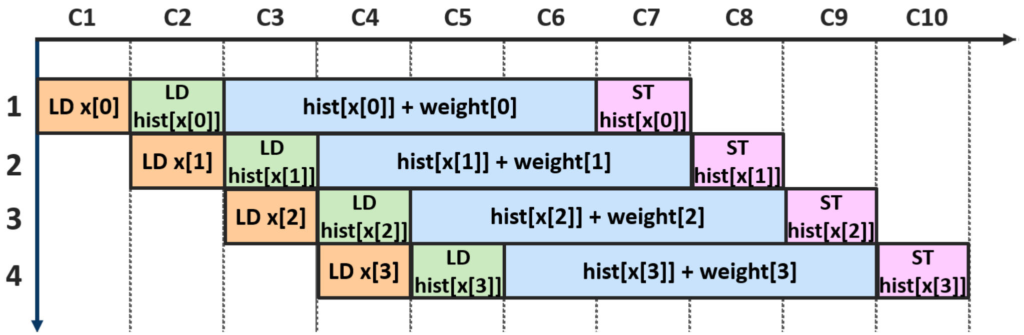


# Optimizing Performance





# Optimizing Performance



C1 C2 C3 C4 C5 C6 C7 C8 C9 C10

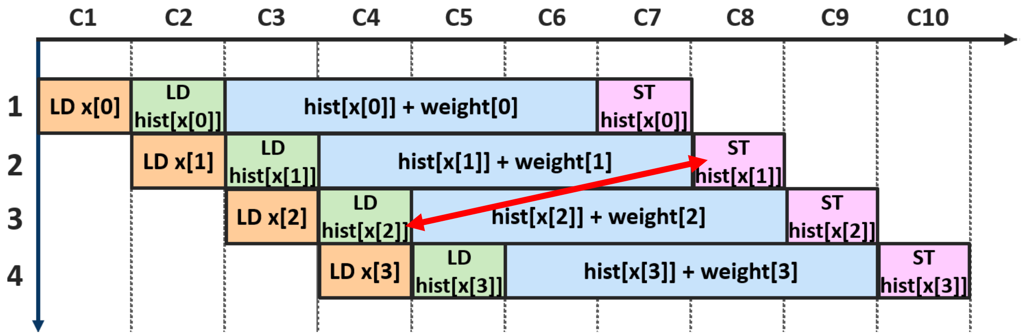
1 LD x[0] | LD hist[x[0]] | hist[x[0]] + weight[0] | ST hist[x[0]]

2 LD x[1] | LD hist[x[1]] | hist[x[1]] + weight[1] | ST hist[x[1]]

3 LD x[2] | LD hist[x[2]] | hist[x[2]] + weight[2] | ST hist[x[2]]

4 LD x[3] | LD hist[x[3]] | hist[x[3]] + weight[3] | ST hist[x[3]]

# Optimizing Performance



RAW dependency
not honored!

**What about memory?**

# Part 3 Outline

✓ What traditional HLS does not do well

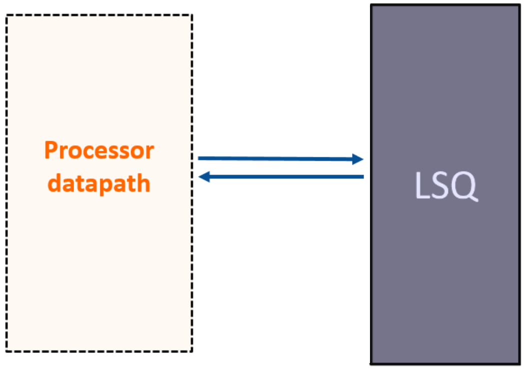
✓ Synthesis of dataflow circuits

✓ Buffers and performance

## The problem with memory

# We Need a Load-Store Queue (LSQ)!

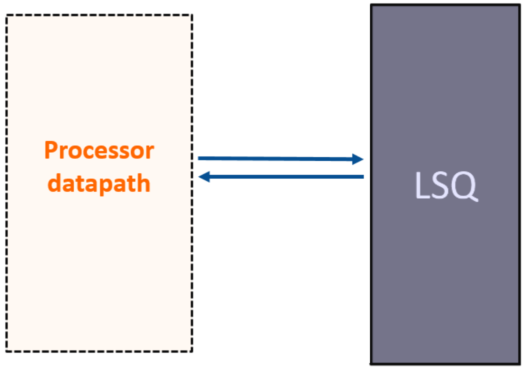
- Traditional processor LSQs allocate memory instructions **in program order**



```
loop: lw $t1, 0($t0)
      lw $t2, 0($t1)
      mul $t2, $t2, $t3
      sw $t2, 0($t0)
      addi $t1, $t1, 4
      bne $t5, $t1, loop
```

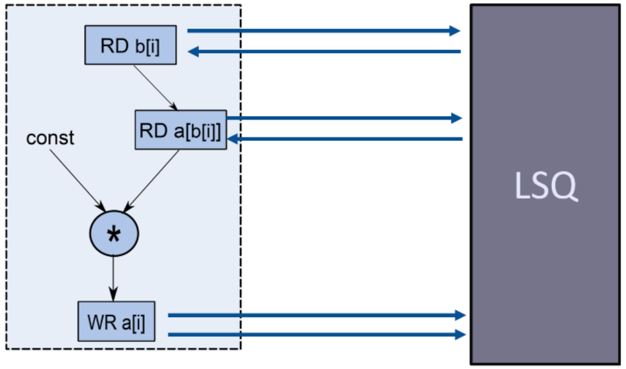
# We Need a Load-Store Queue (LSQ)!

- Traditional processor LSQs allocate memory instructions **in program order**



```
loop: lw $t1, 0($t0)
      lw $t2, 0($t1)
      mul $t2, $t2, $t3
      sw $t2, 0($t0)
      addi $t1, $t1, 4
      bne $t5, $t1, loop
```

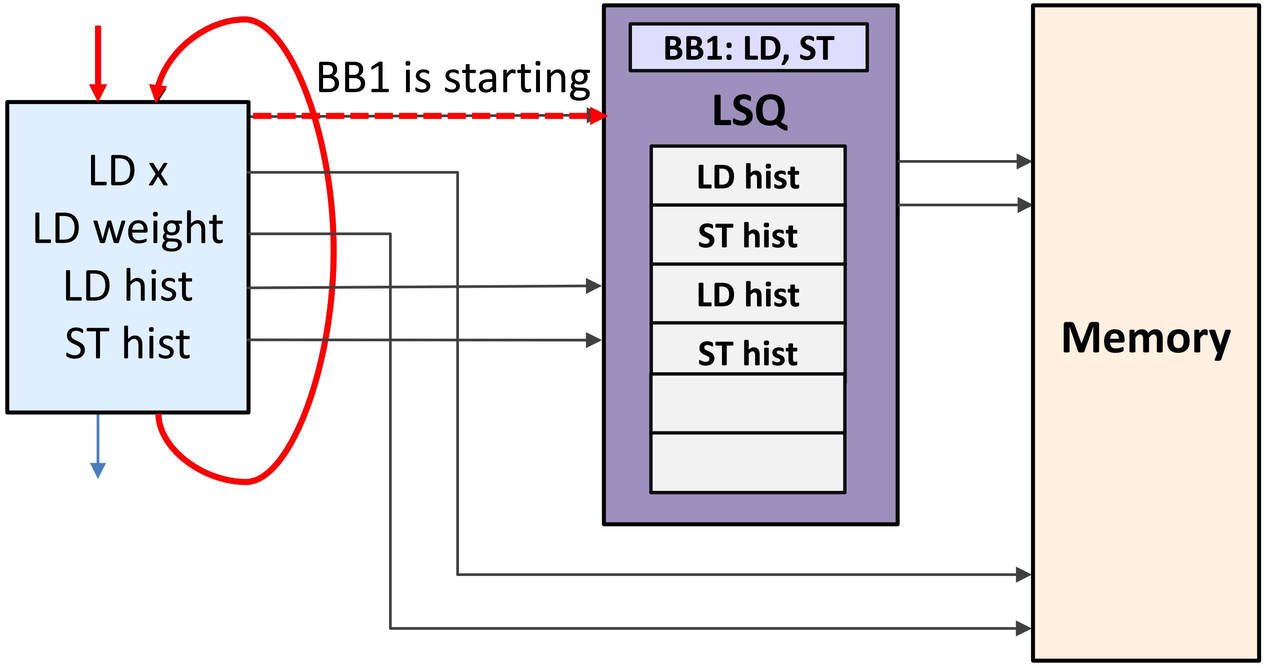
- Dataflow circuits have **no notion of program order**



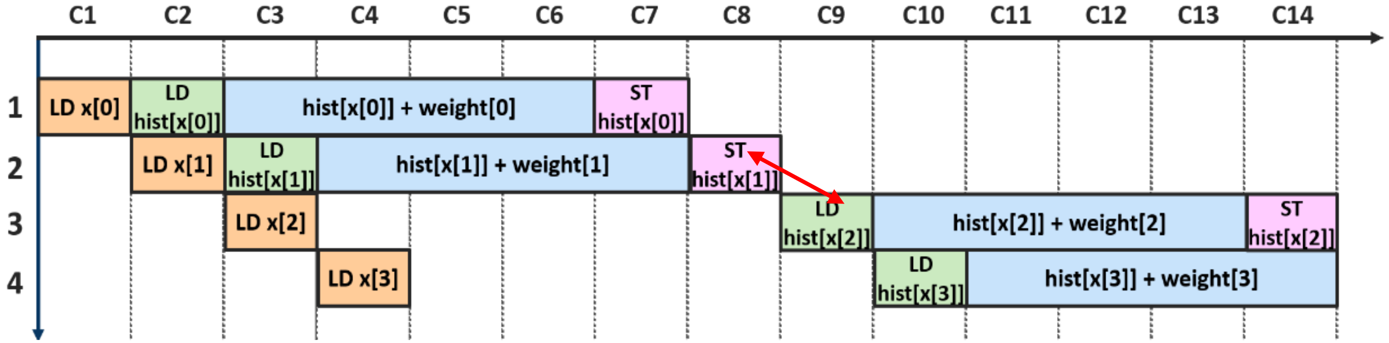
**How to supply program order to the LSQ?**

# Basic Idea

- An LSQ for dataflow circuits whose only difference is in the **allocation policy**:
  - **Static knowledge** of memory accesses program order inside each basic block
  - **Dynamic knowledge** of the sequence of basic blocks **from the dataflow circuit**



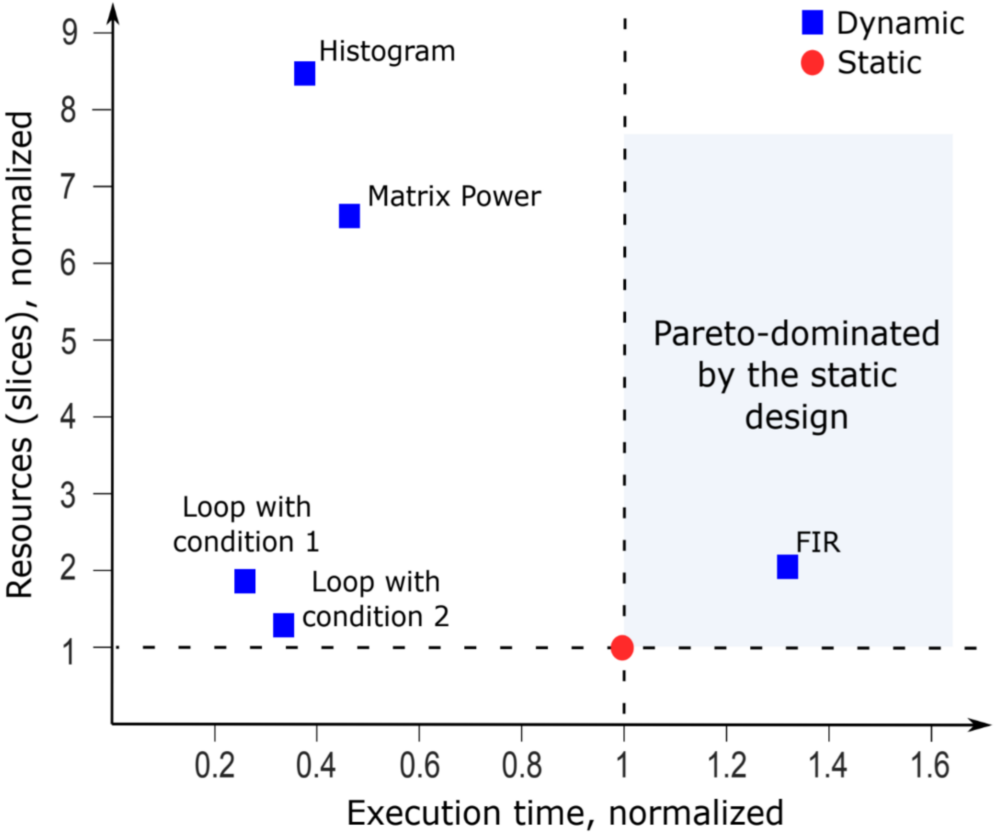
# Dataflow Circuit with the LSQ



High-throughput pipeline with memory dependencies honored

# Experimental Results

- Resource utilization and execution time of the dataflow designs, **normalized to the corresponding static designs** produced by Vivado HLS.



# Part 3 Outline

- ✓ What traditional HLS does not do well
- ✓ Synthesis of dataflow circuits
- ✓ Buffers and performance
- ✓ The problem with memory

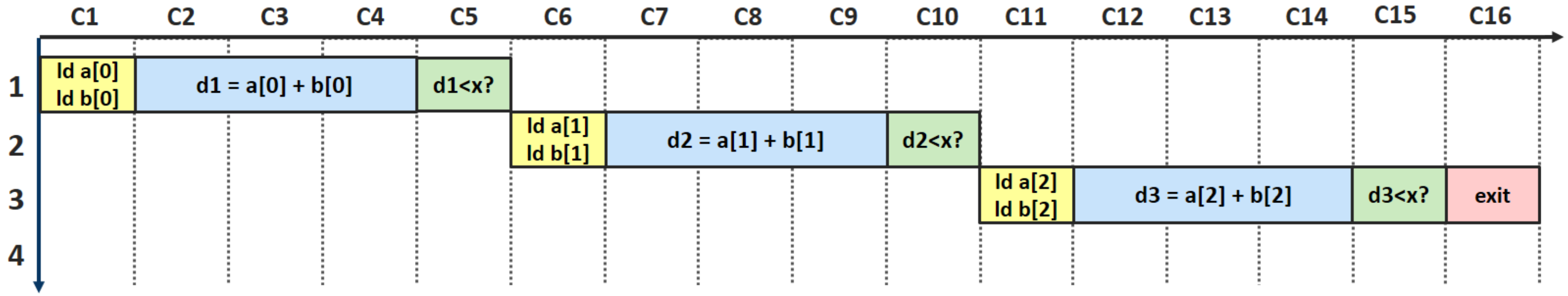
## Conquering new grounds with speculation

# 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



C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15 C16

1: ld a[0] ld b[0] | d1 = a[0] + b[0] | d1<x?

2: ld a[1] ld b[1] | d2 = a[1] + b[1] | d2<x?

3: ld a[2] ld b[2] | d3 = a[2] + b[2] | d3<x? | exit

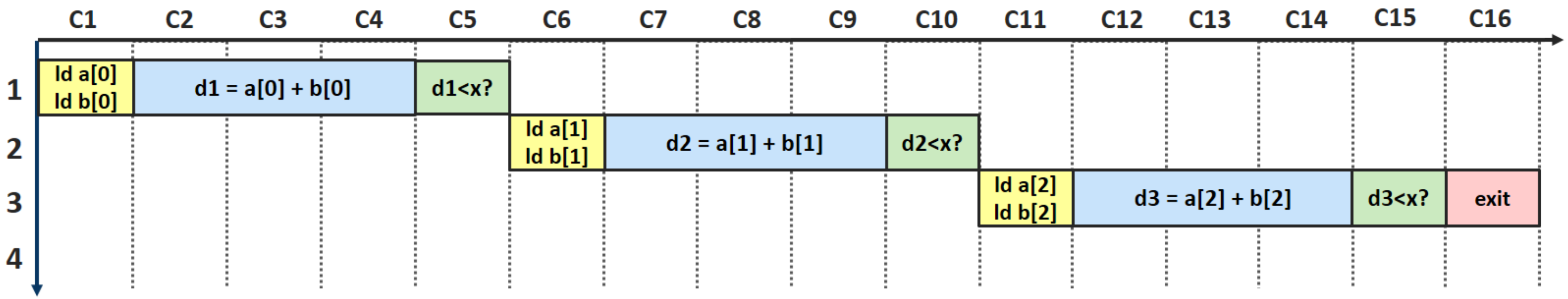
4

# Nonspeculative vs. Speculative System

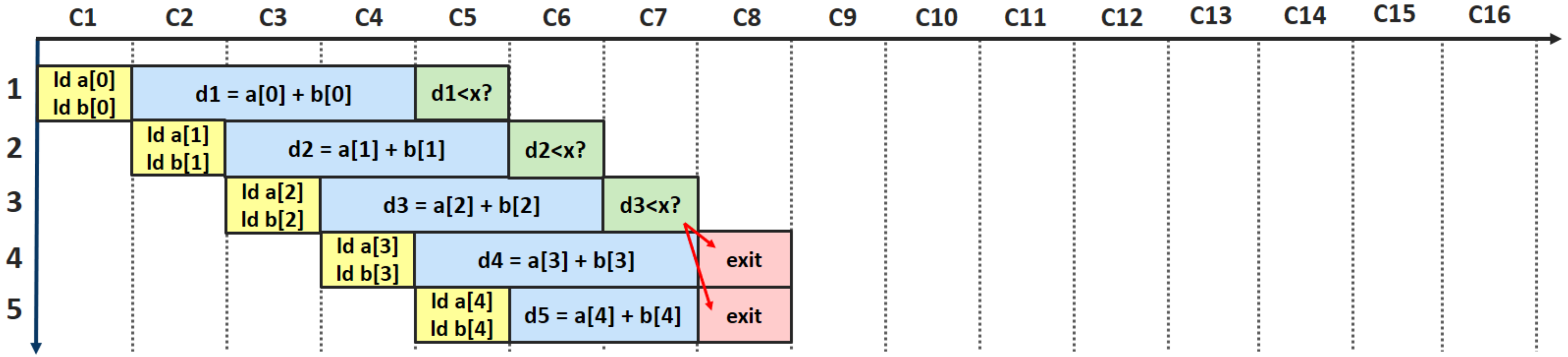
```
float d=0.0; x=100.0; int i=0;

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
```

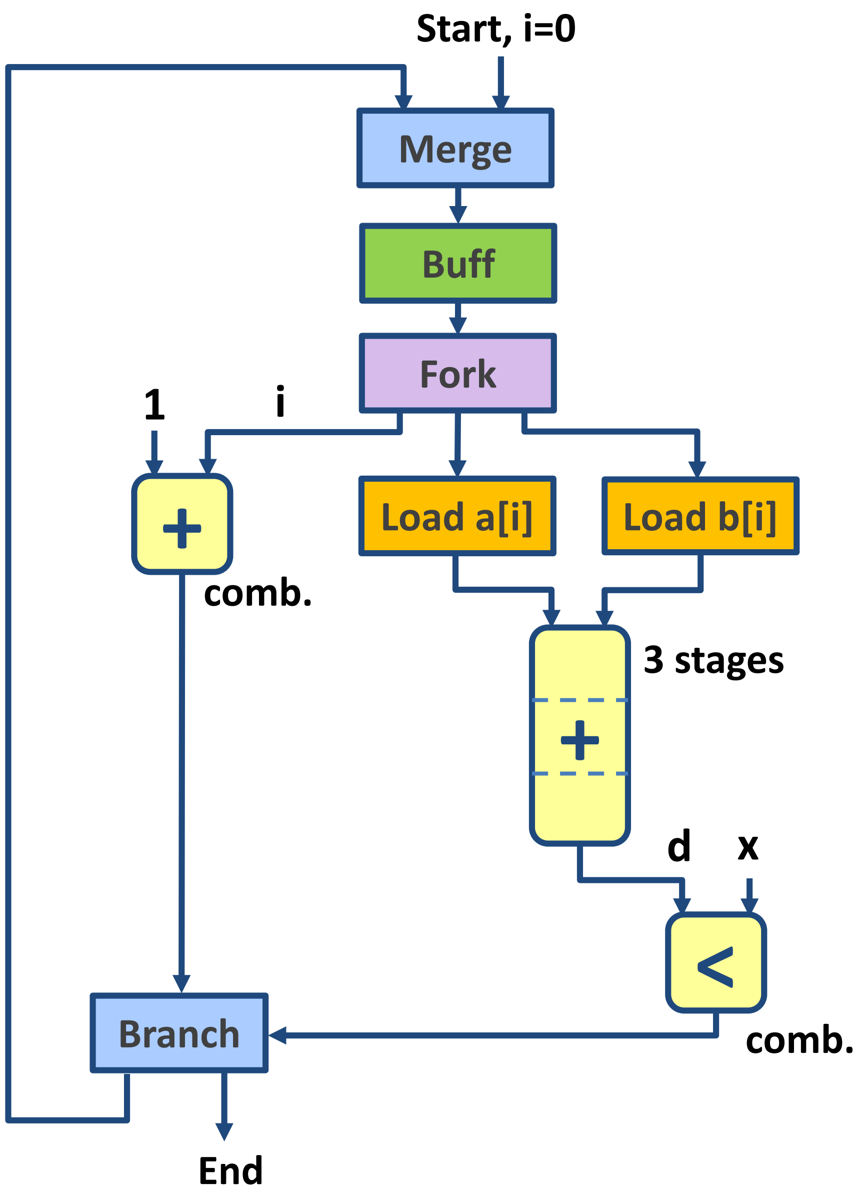
Nonspeculative schedule



Speculative schedule

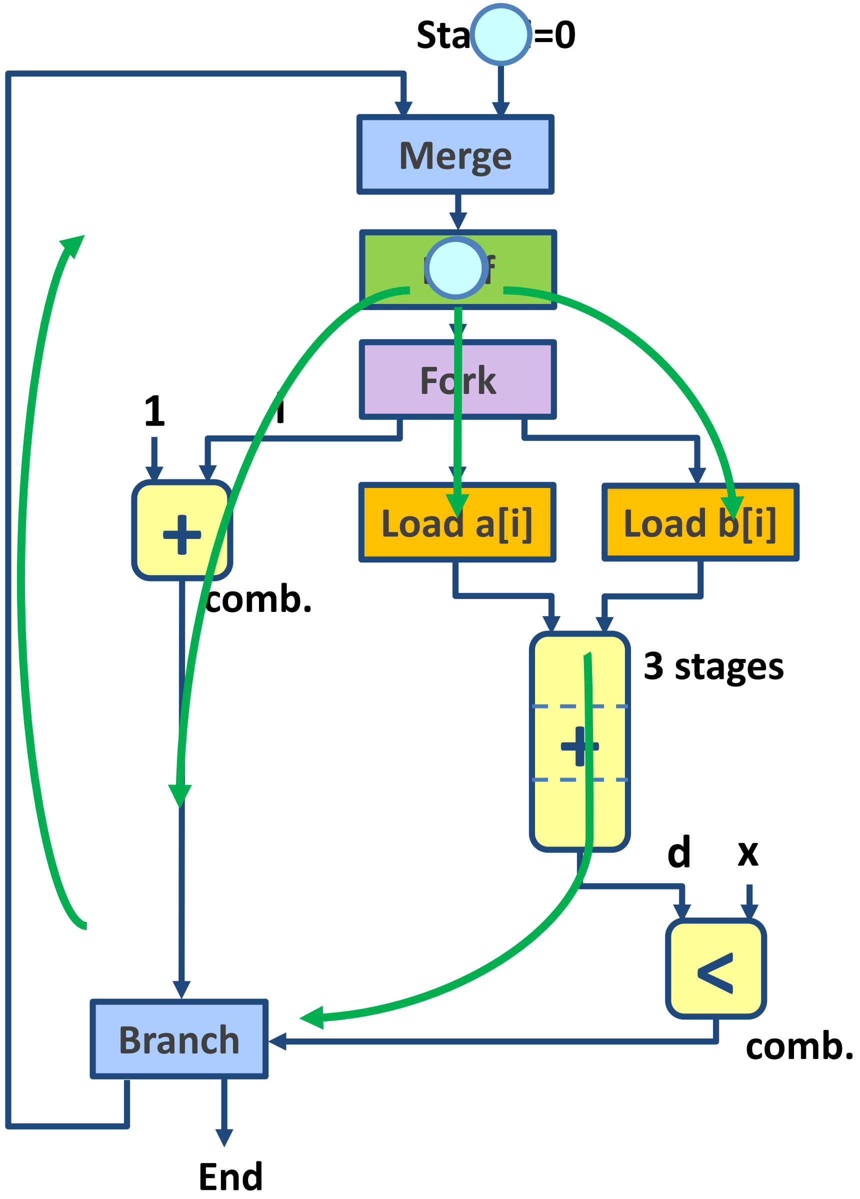


# Nonspeculative Dataflow Circuit



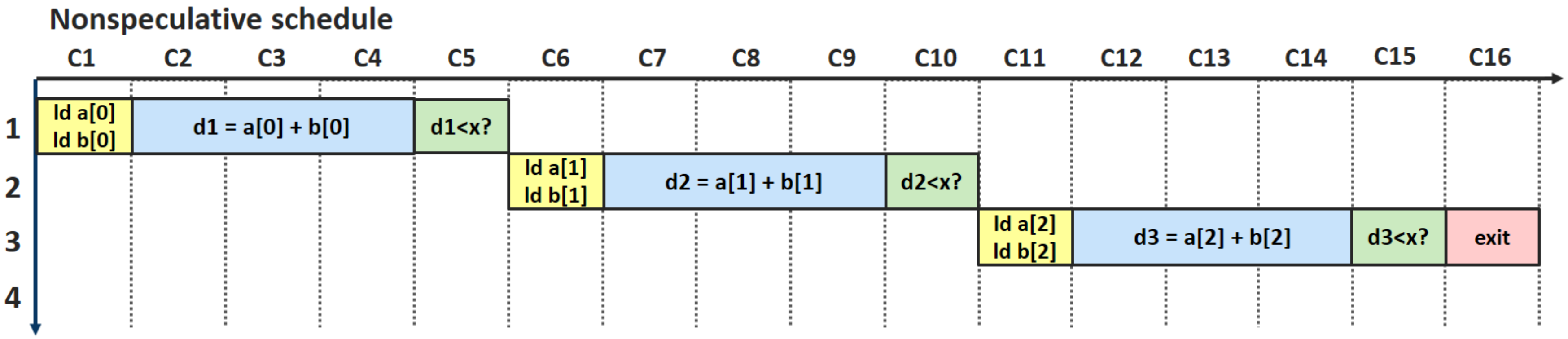
```
float d=0.0; x=100.0; int i=0;

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



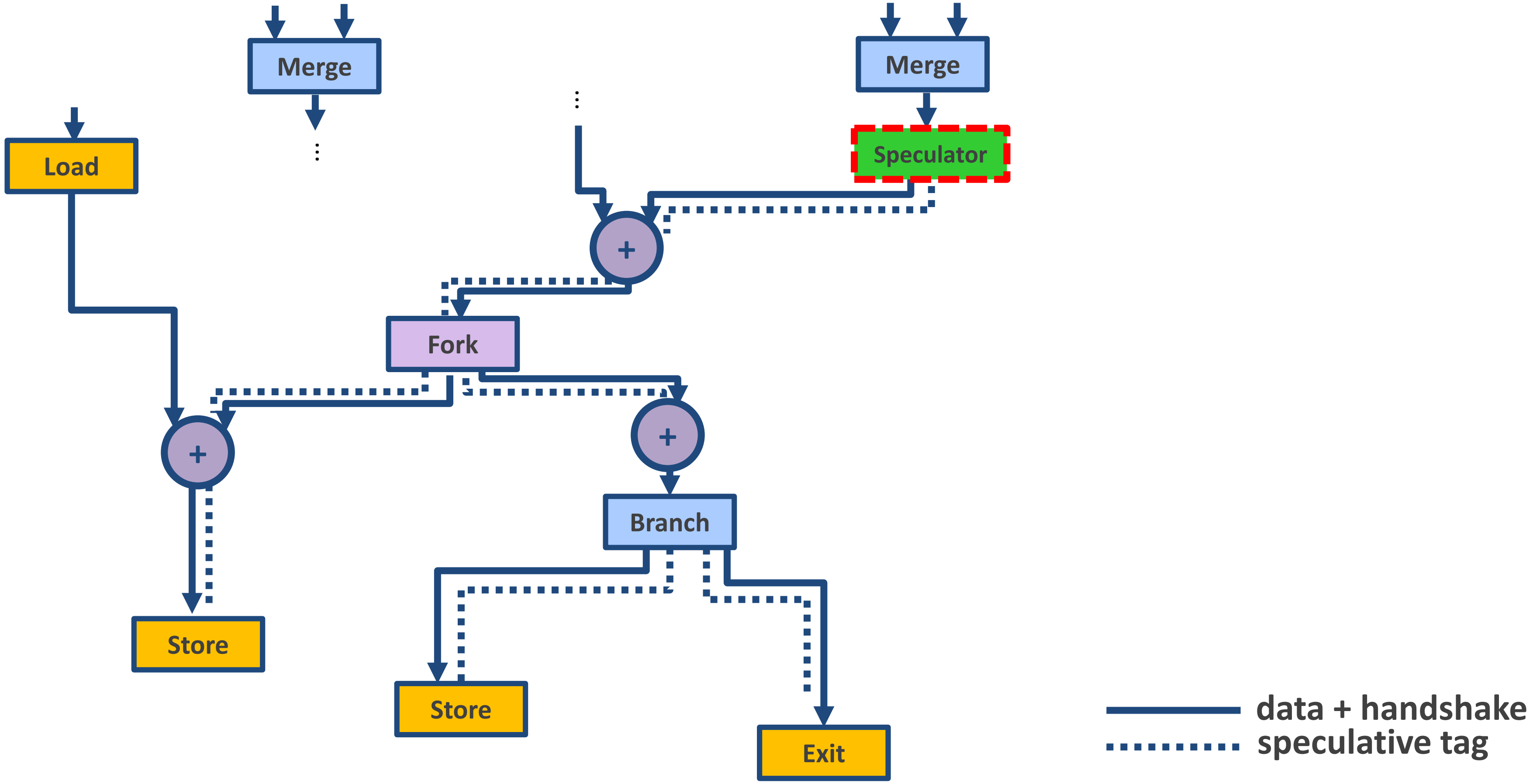
Nonspeculative schedule

|  | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | C11 | C12 | C13 | C14 | C15 | C16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ld a[0]<br>ld b[0] | d1 = a[0] + b[0] |  |  | d1<x? |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  | ld a[1]<br>ld b[1] | d2 = a[1] + b[1] |  |  | d2<x? |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  | ld a[2]<br>ld b[2] | d3 = a[2] + b[2] |  |  | d3<x? | exit |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Long control flow decision prevents pipelining**

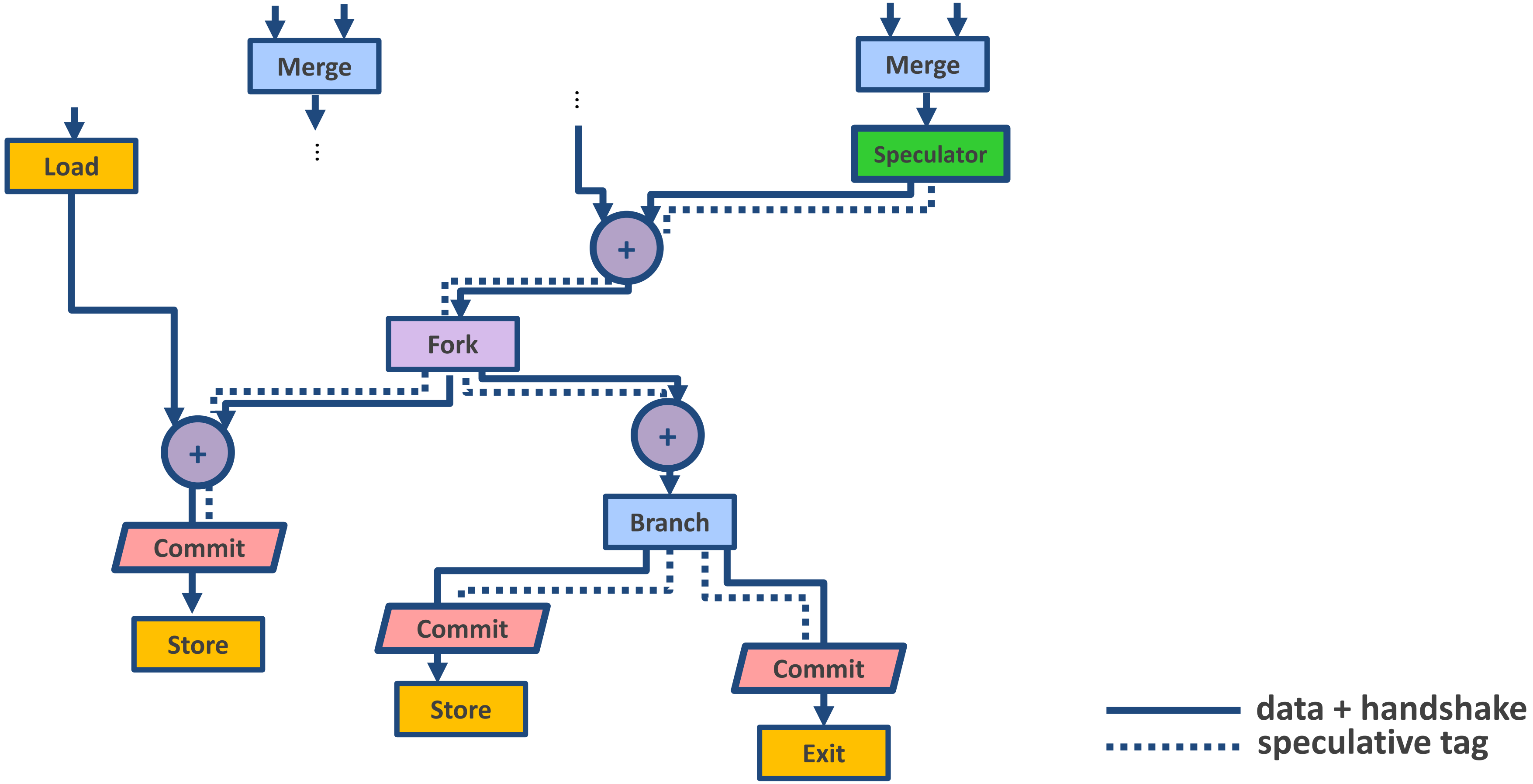
# Speculation in Dataflow Circuits

- 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



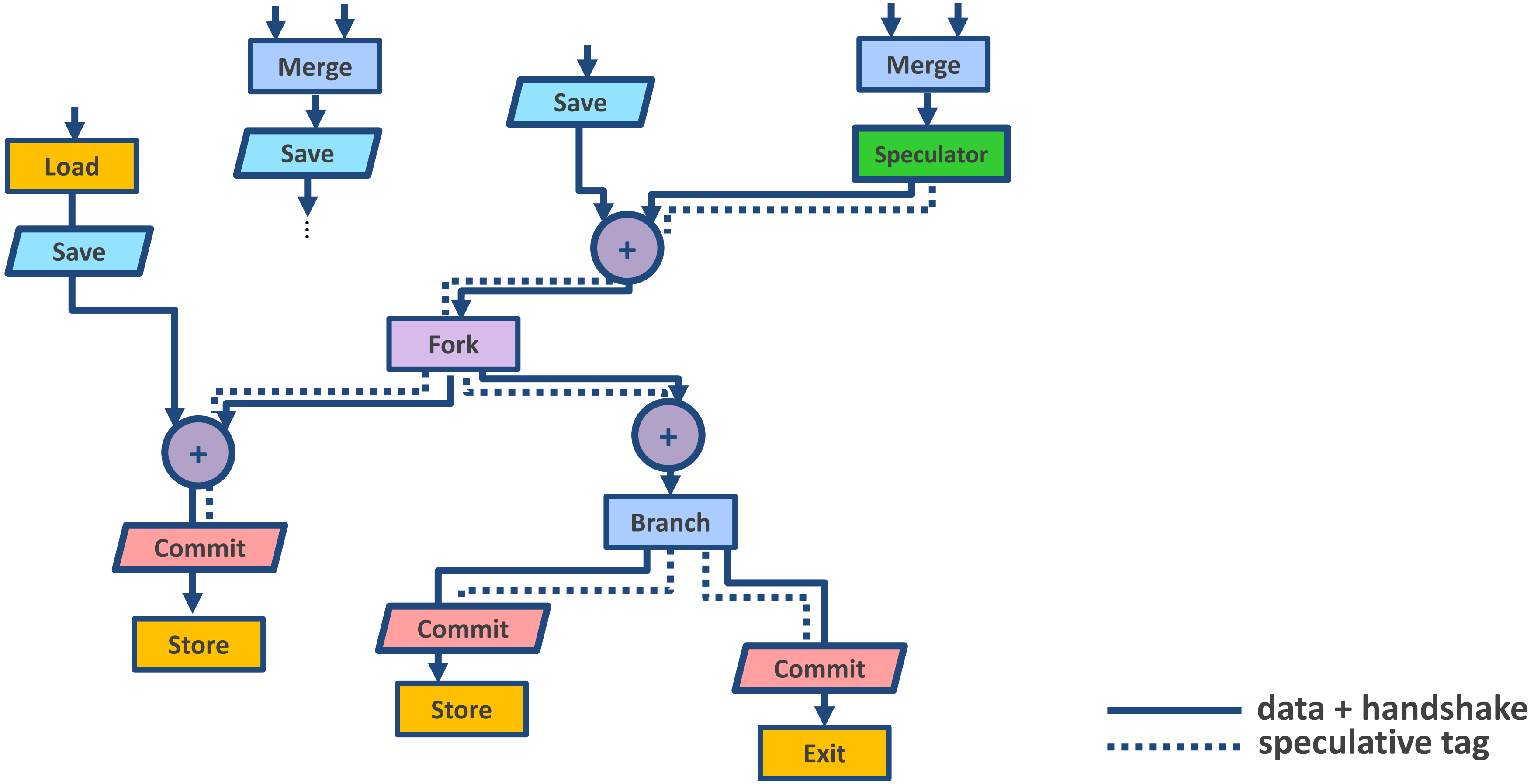
# Speculation in Dataflow Circuits

- 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



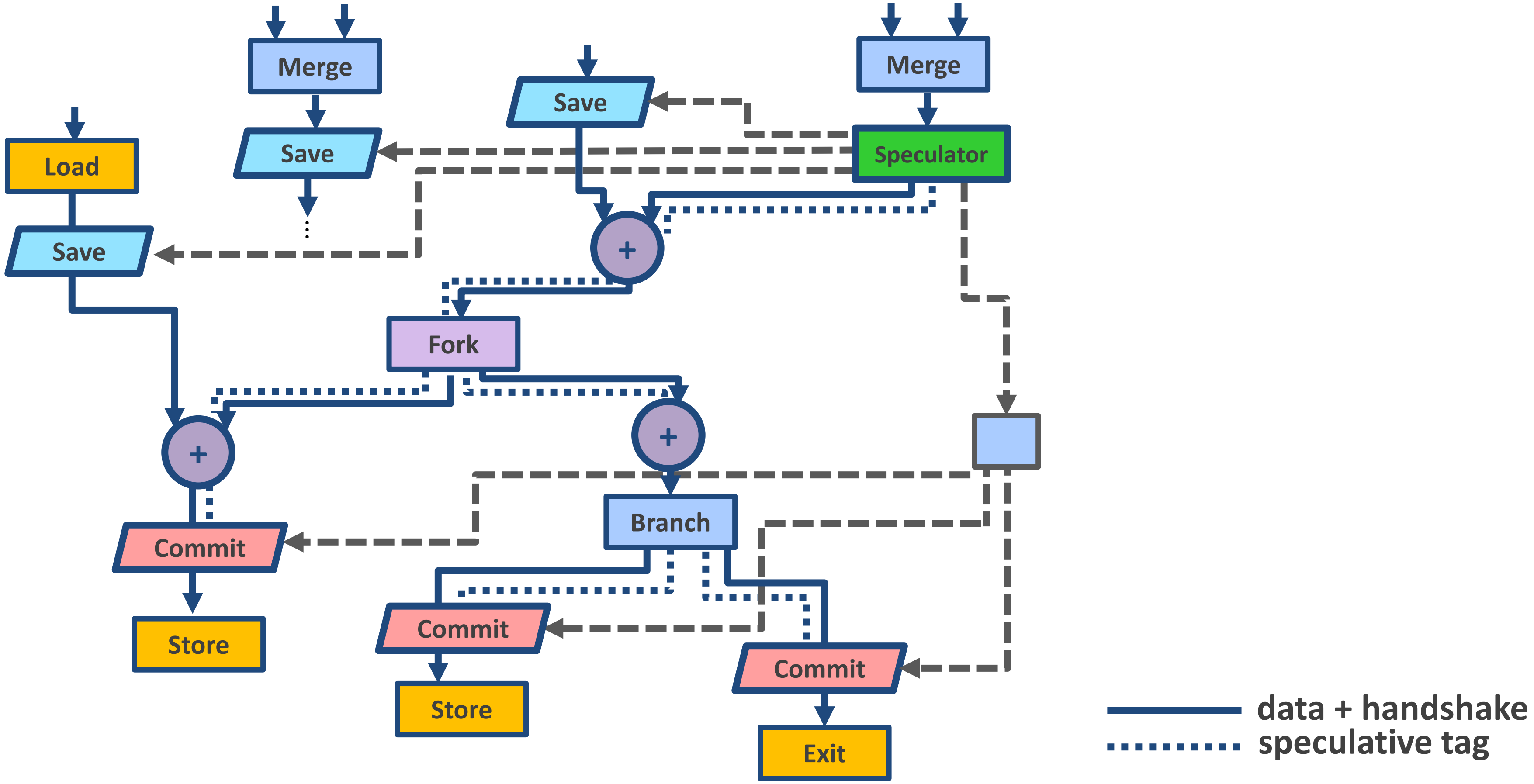
# Speculation in Dataflow Circuits

- 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



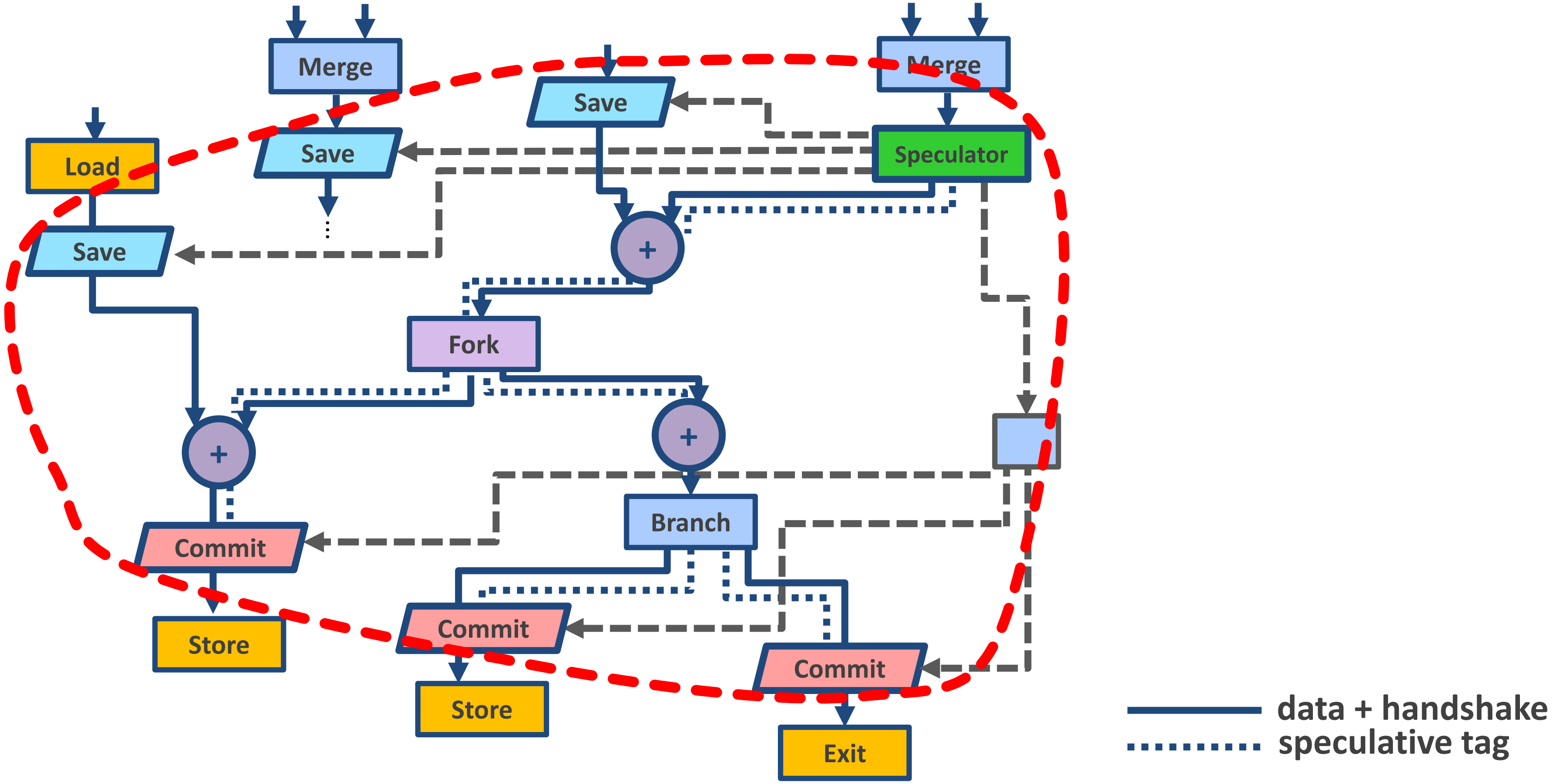
# Speculation in Dataflow Circuits

- 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



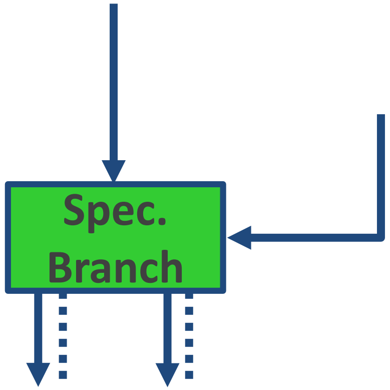
# Speculation in Dataflow Circuits

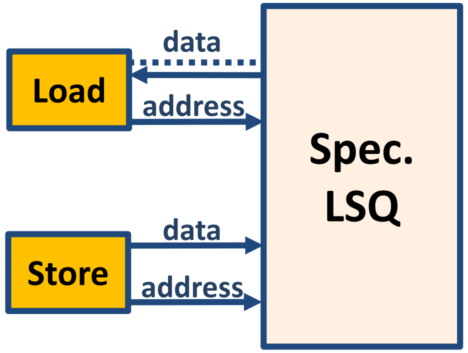
- Contain speculation in a region of the circuit delimited by special components
  - Issue speculative tokens (pieces of data which might or might not be correct)
  - Squash and replay in case of misspeculation



# Components for Speculation

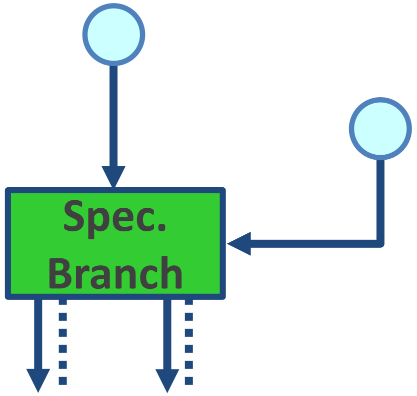
- Speculator
  - Dataflow component which can, besides its standard functionality, also inject tokens before receiving any at its input(s)
  - Branch Speculator, LSQ





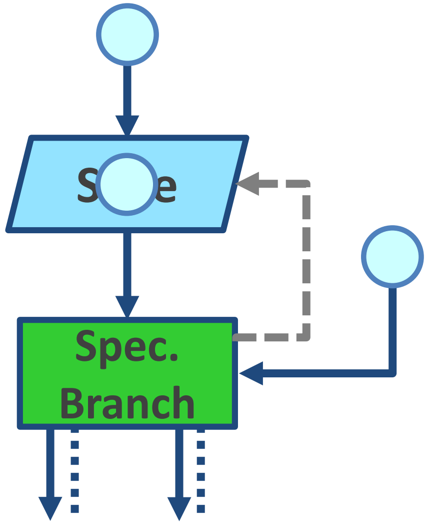
# Components for Speculation

- Speculator
  - Dataflow component which can, besides its standard functionality, also inject tokens before receiving any at its input(s)
  - Branch Speculator, LSQ



# Components for Speculation

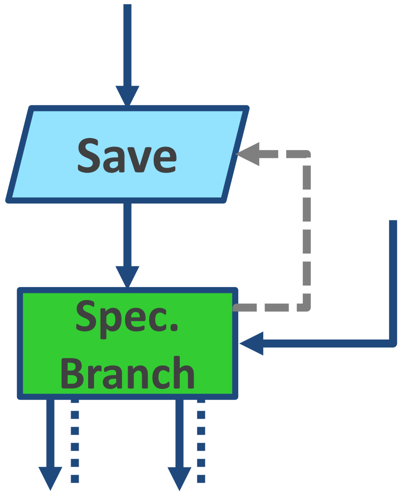
- Save units
  - Input boundary of the speculative region
  - Reissues when previous computation is squashed

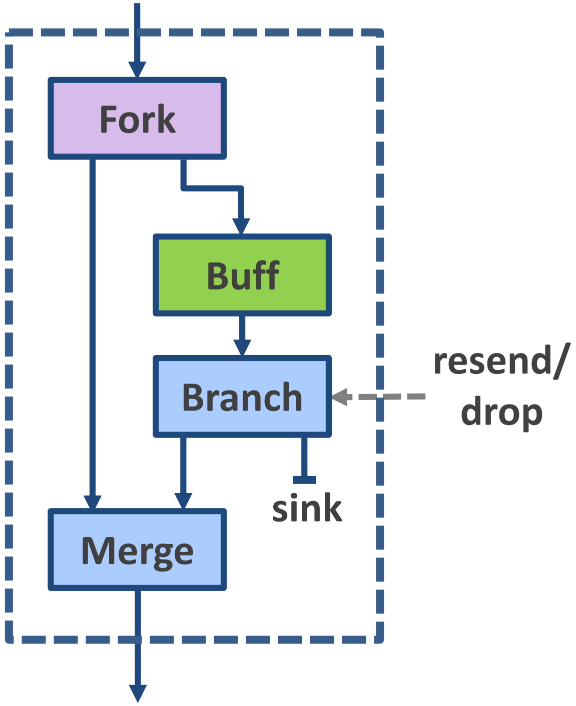


**Save a copy of all regular tokens which may become speculative**

# Components for Speculation

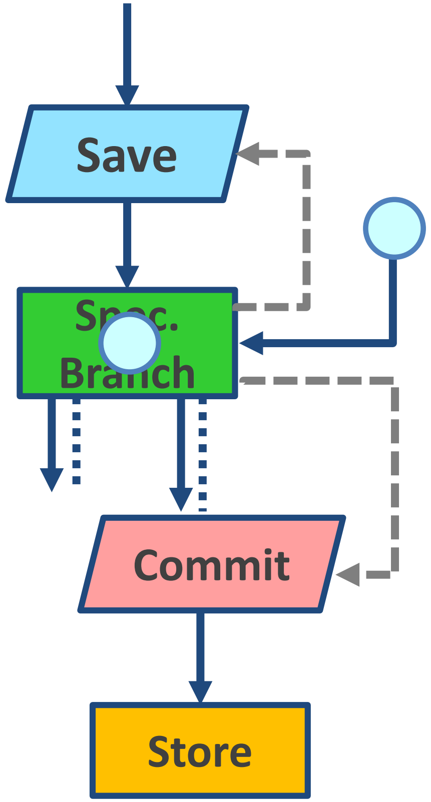
- Save units
  - Input boundary of the speculative region
  - Reissues when previous computation is squashed





# Components for Speculation

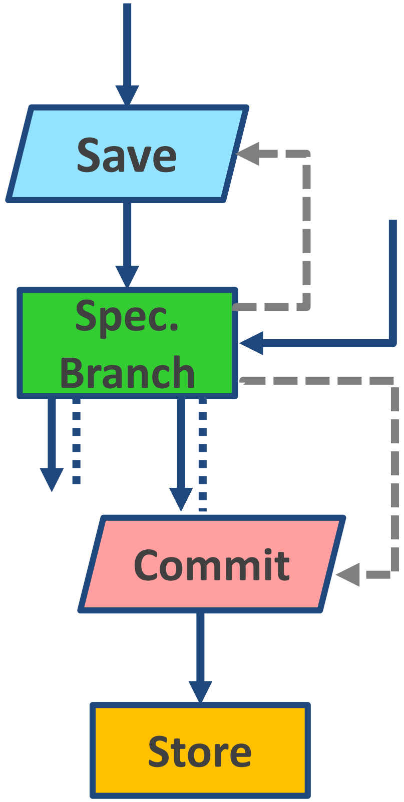
- Commit units
  - Output boundary of the speculative region
  - Propagate speculative tokens that turn out to be correct, squash misspeculated data

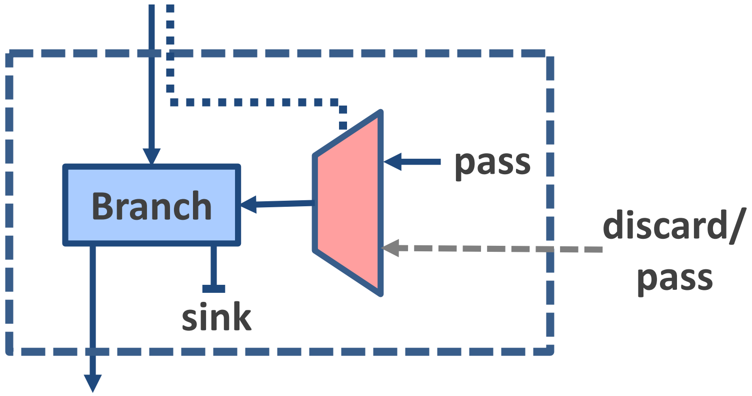


Propagate further speculative results which turn out to be correct

# Components for Speculation

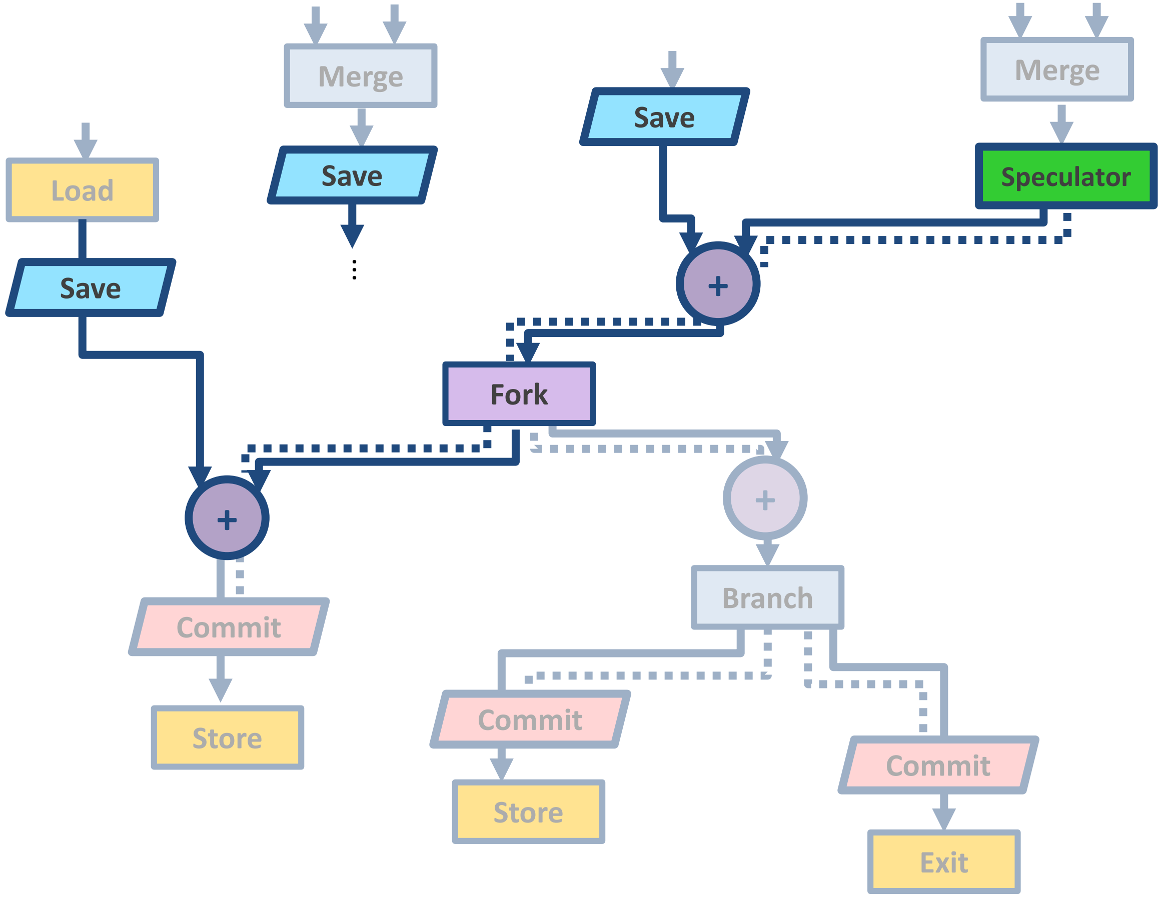
- Commit units
  - Output boundary of the speculative region
  - Propagate speculative tokens that turn out to be correct, squash misspeculated data





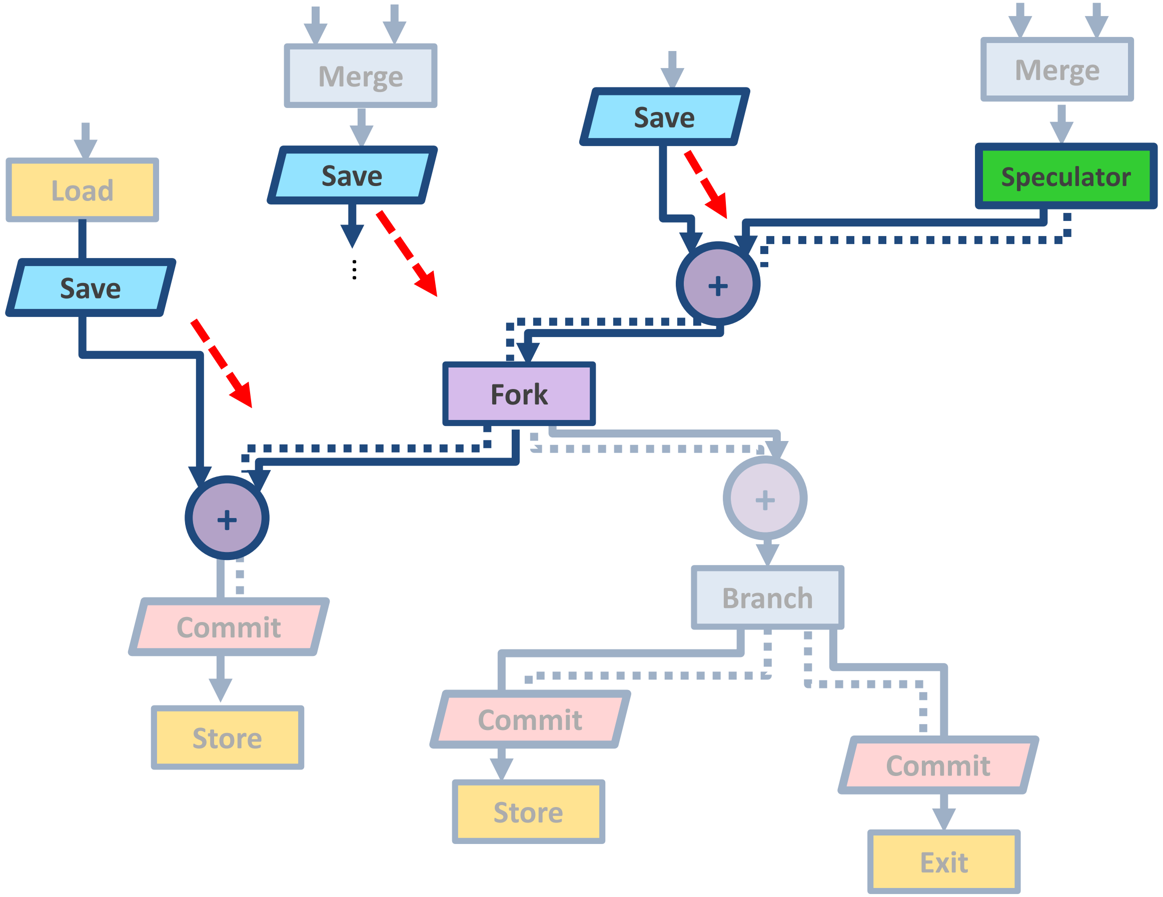
# Placing the Components for Speculation

- Save units
  - On each path to any component that could combine the token with a speculative
  - As close as possible to the paths carrying speculative tokens



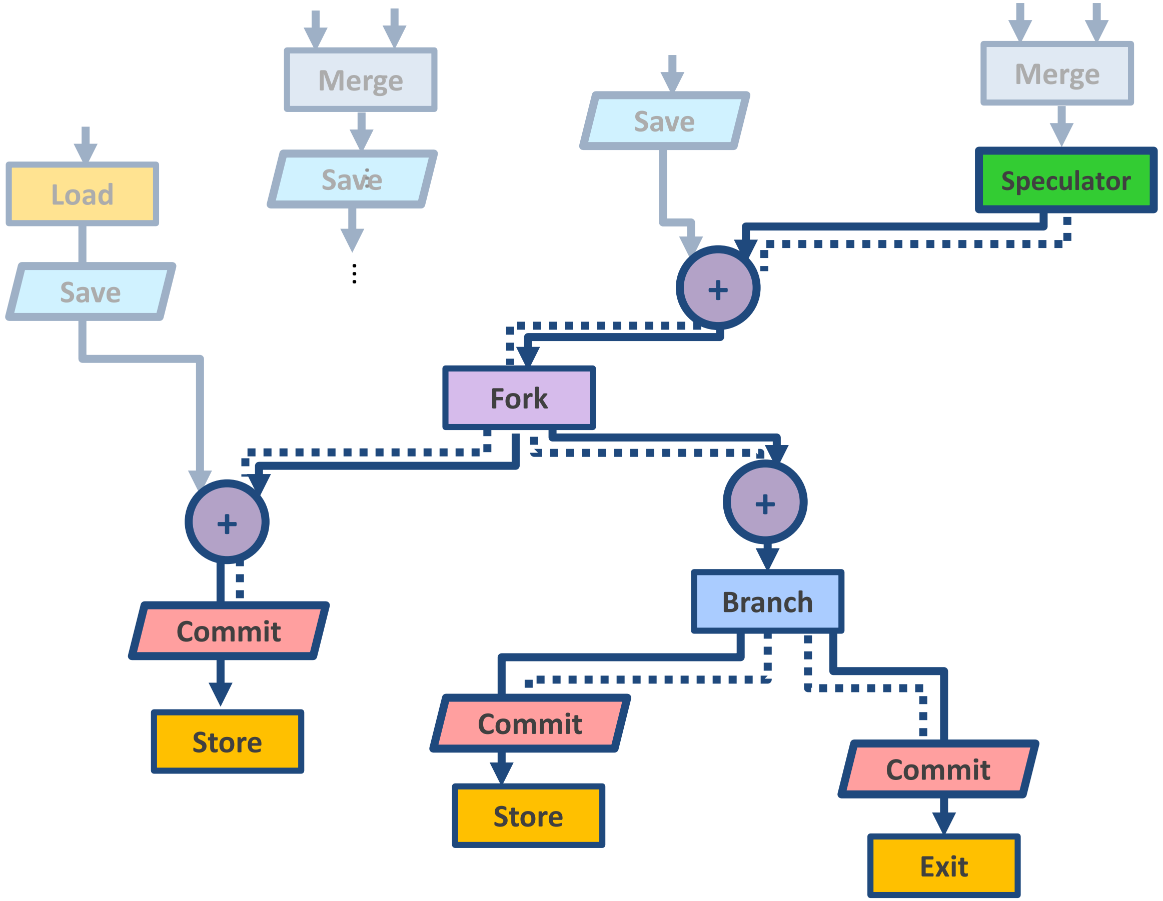
# Placing the Components for Speculation

- Save units
  - On each path to any component that could combine the token with a speculative
  - As close as possible to the paths carrying speculative tokens



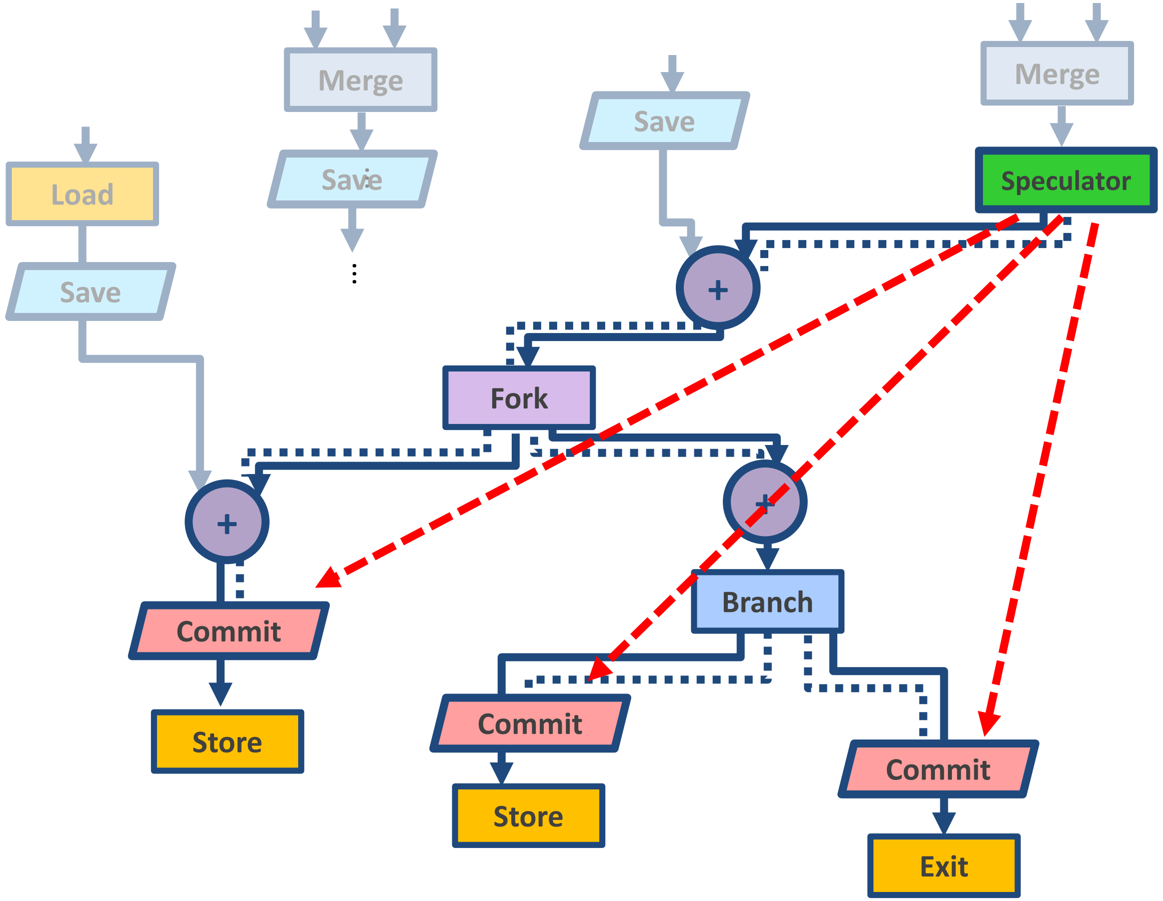
# Placing the Components for Speculation

- Commit units
  - On each path from the Speculator to an exit point, a store unit, or the Speculator
  - As far as possible from the Speculator



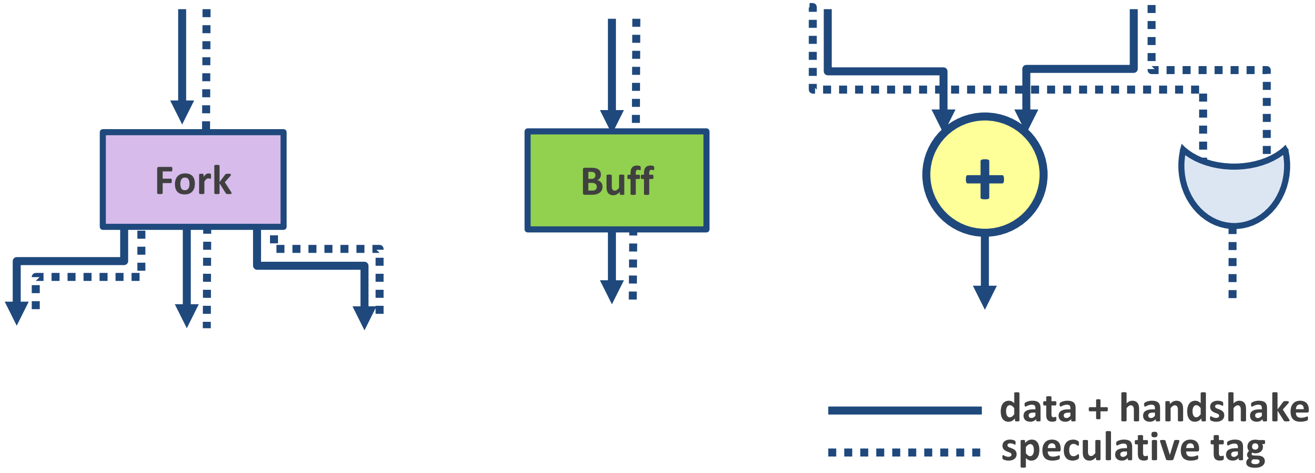
# Placing the Components for Speculation

- Commit units
  - On each path from the Speculator to an exit point, a store unit, or the Speculator
  - As far as possible from the Speculator

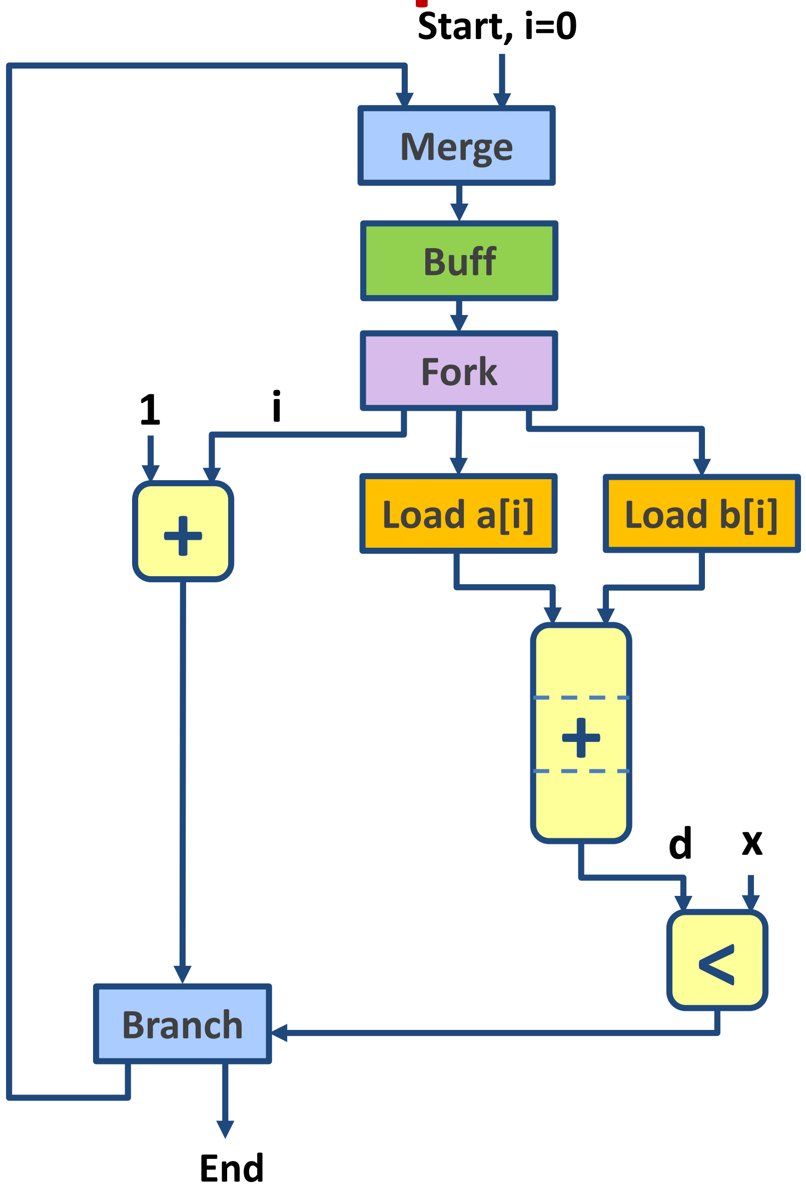


# Speculative Tag

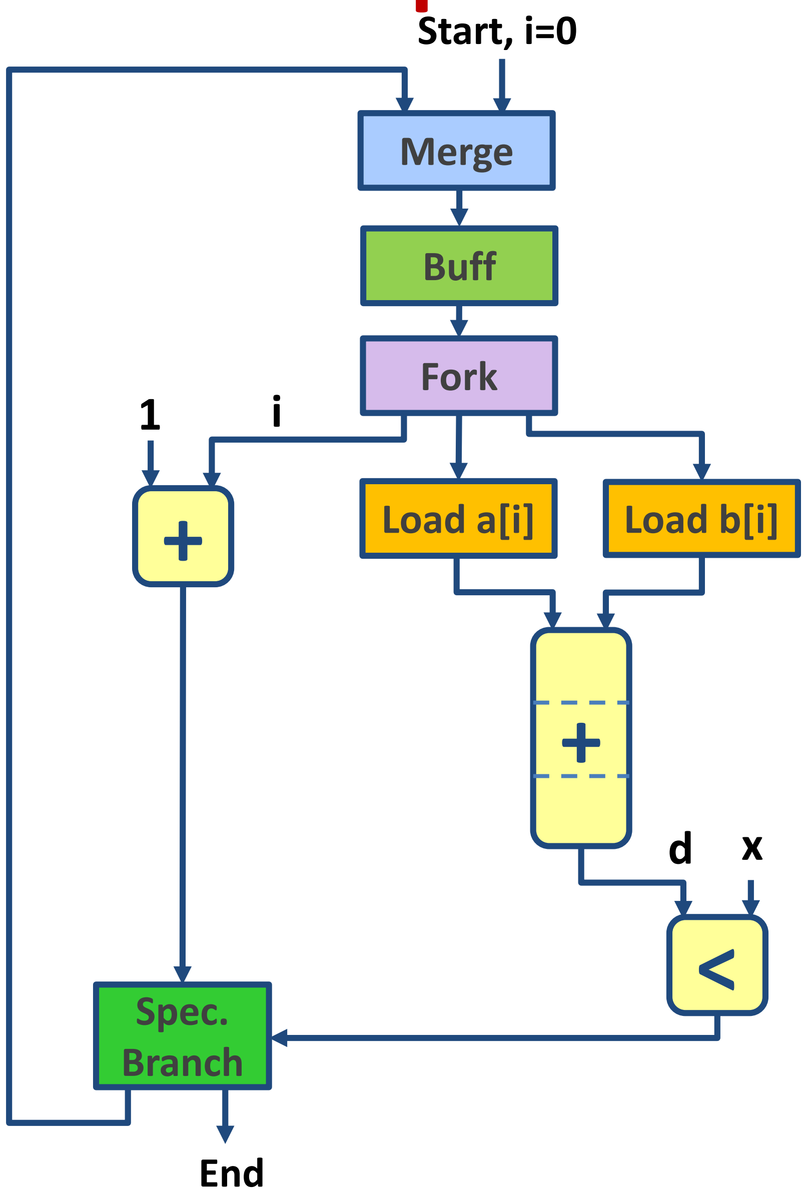
- Extending dataflow components with a speculative tag
  - An additional bit propagated with the data or OR’ed from all inputs



# Speculative Dataflow Circuit

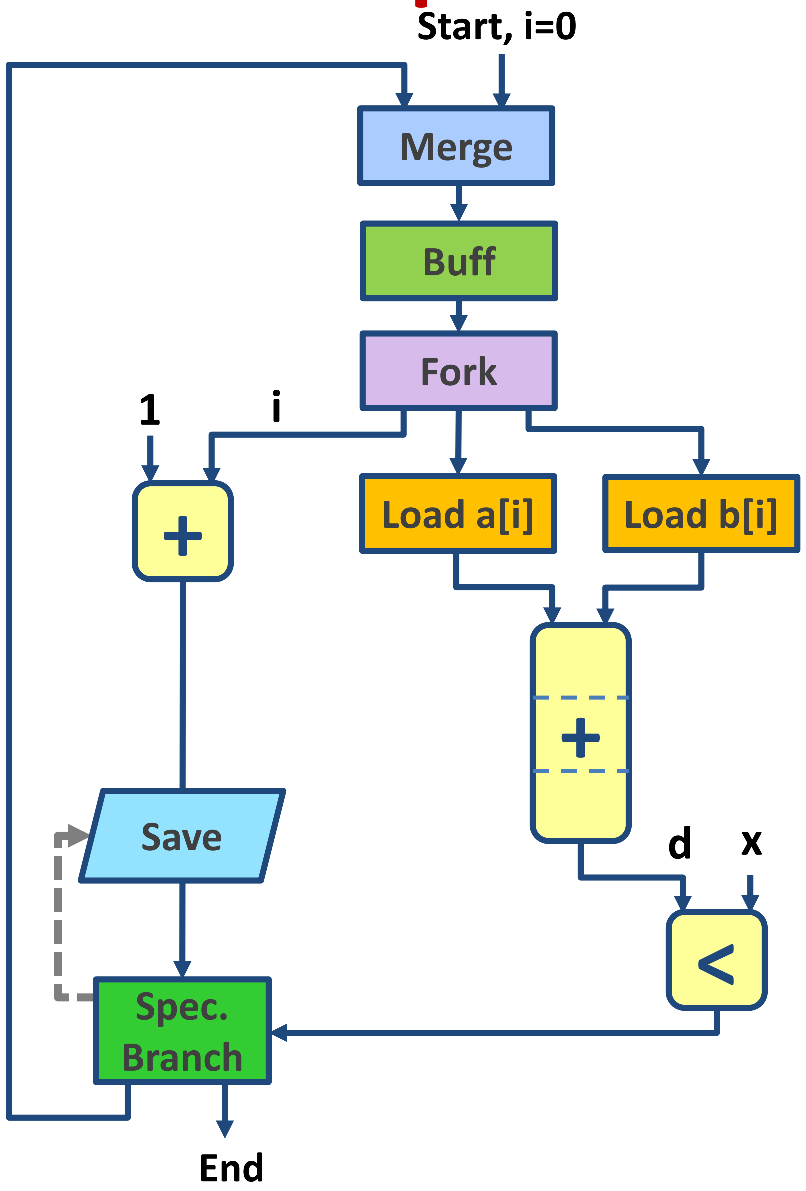


# Speculative Dataflow Circuit



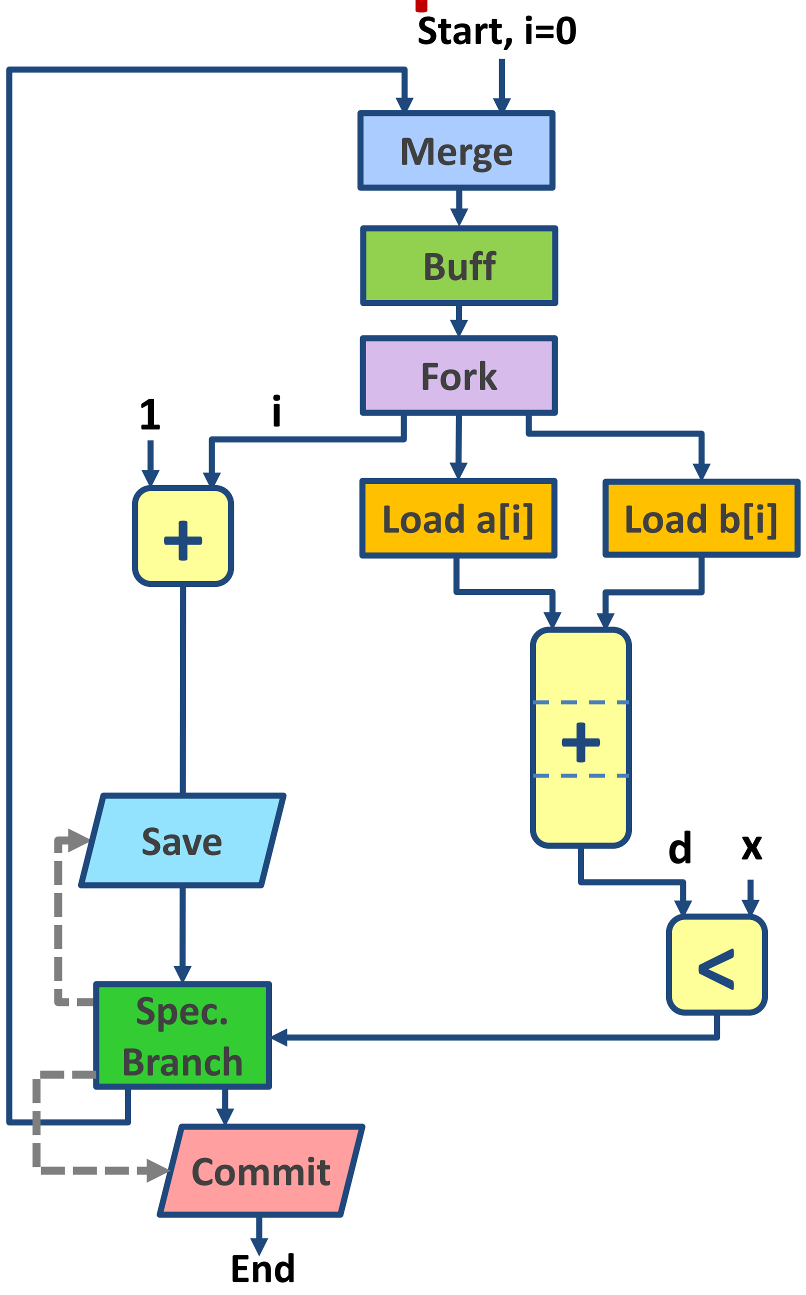
Speculator instead of regular Branch

# Speculative Dataflow Circuit



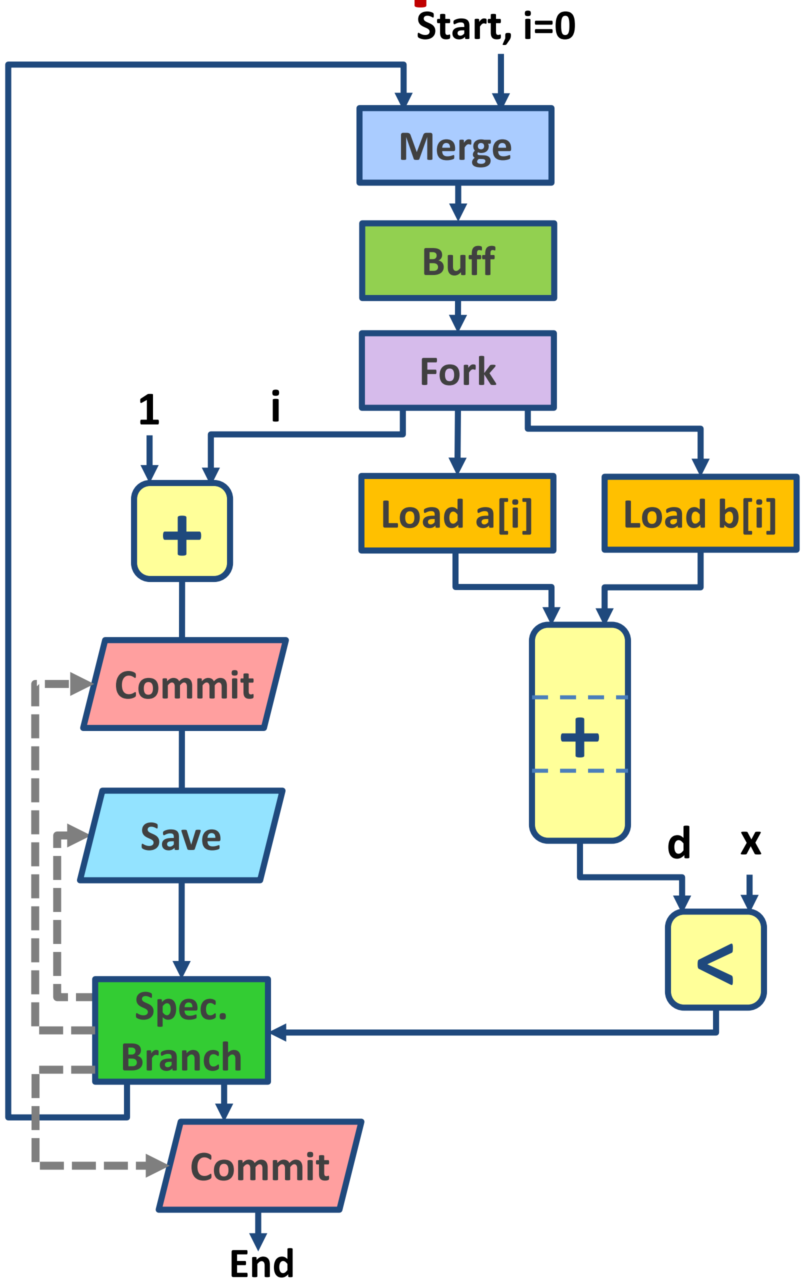
Input boundary:
Save units

# Speculative Dataflow Circuit



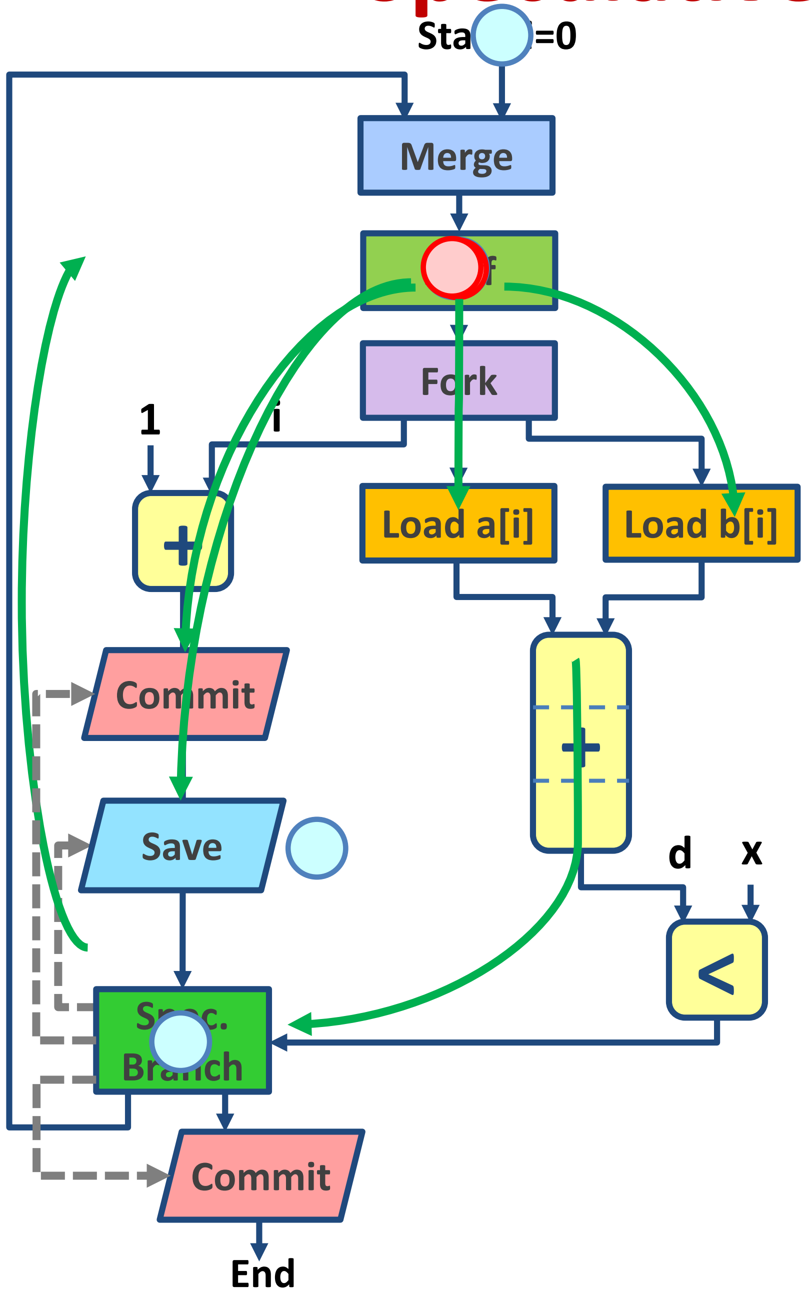
Output boundary:
Commit units

# Speculative Dataflow Circuit

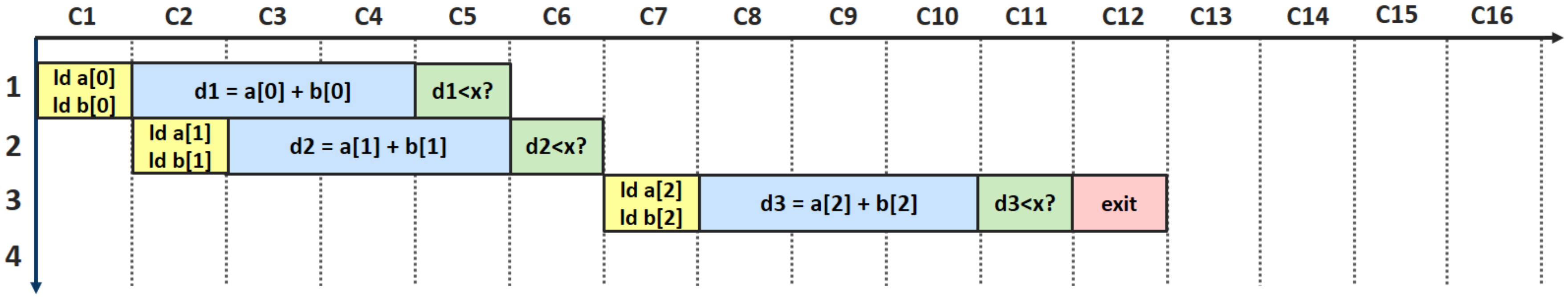


Output boundary:
Commit units

# Speculative Dataflow Circuit



# Speculative Dataflow Circuit



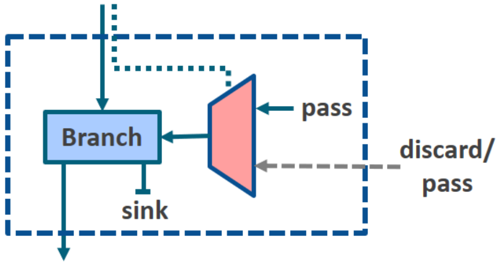
**Single speculation at a time**
**cannot achieve maximum parallelism**

# Increasing Performance

- Merging the Save and Commit unit on cyclic paths

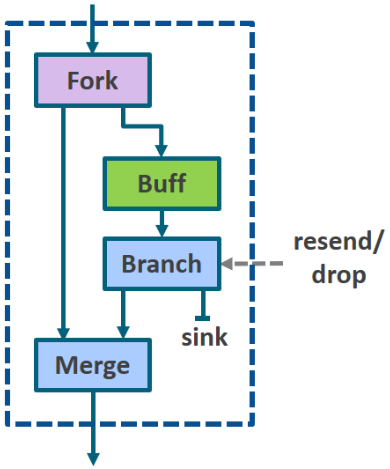
**Commit unit:**

- Stalls speculative tokens
- Discards misspeculated tokens



**Save unit:**

- Saves and reissues tokens

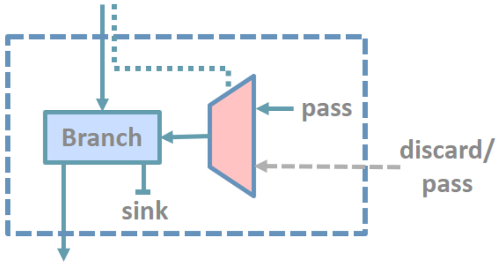


# Increasing Performance

- Merging the Save and Commit unit on cyclic paths

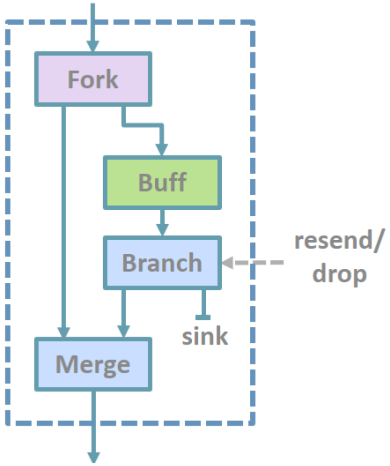
**Commit unit:**

- Stalls speculative tokens
- Discards misspeculated tokens



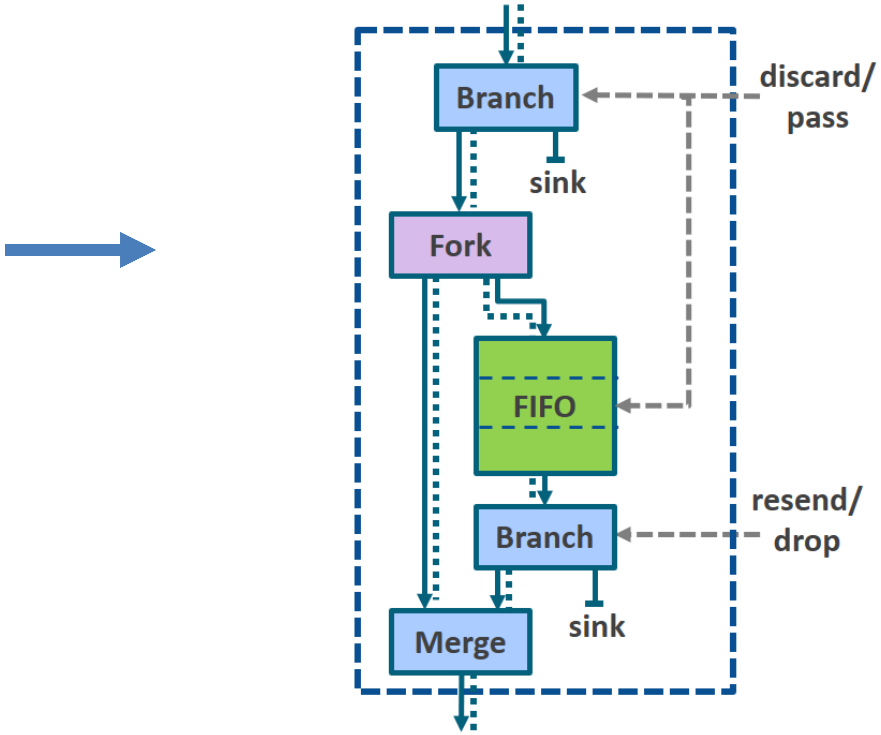
**Save unit:**

- Saves and reissues tokens



**Save-Commit unit:**

- Lets speculative tokens pass
- Discards misspeculated tokens
- Saves and reissues tokens

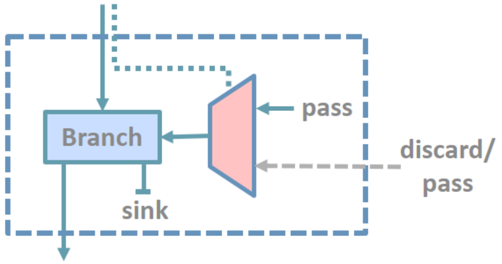


# Increasing Performance

- Merging the Save and Commit unit on cyclic paths

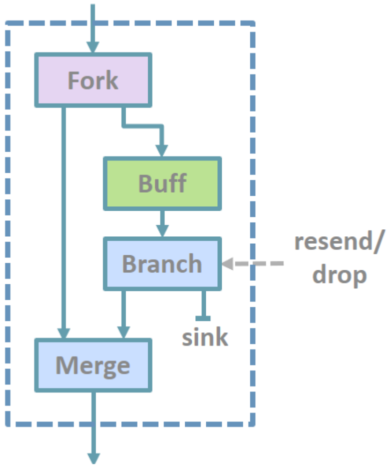
**Commit unit:**

- Stalls speculative tokens
- Discards misspeculated tokens



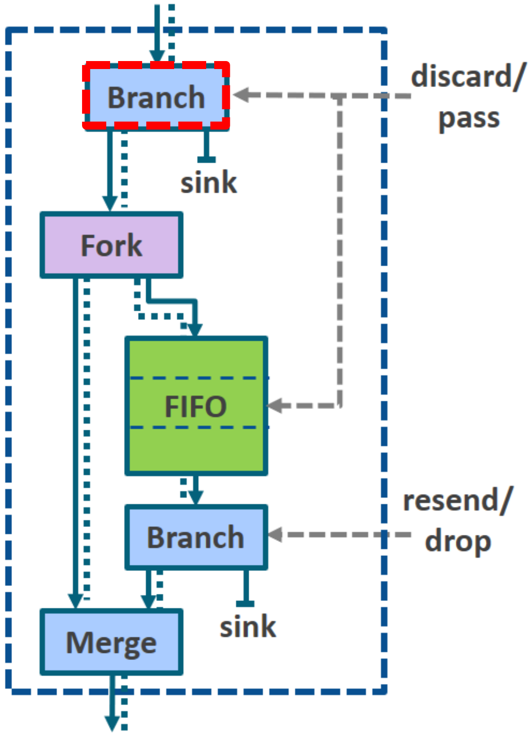
**Save unit:**

- Saves and reissues tokens



**Save-Commit unit:**

- Lets speculative tokens pass
- Discards misspeculated tokens
- Saves and reissues tokens

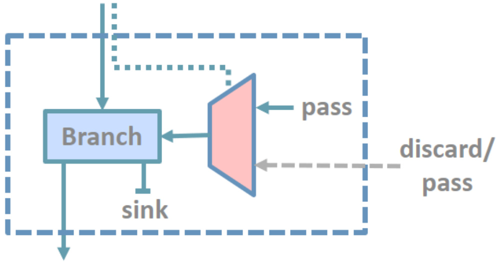


# Increasing Performance

- Merging the Save and Commit unit on cyclic paths

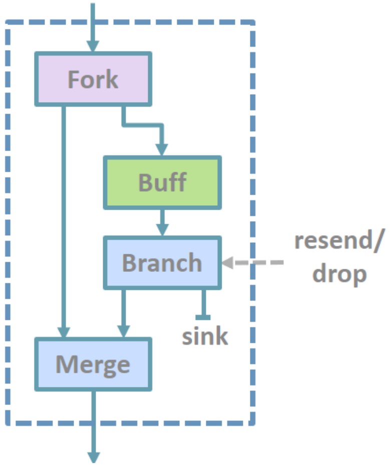
**Commit unit:**

- Stalls speculative tokens
- Discards misspeculated tokens



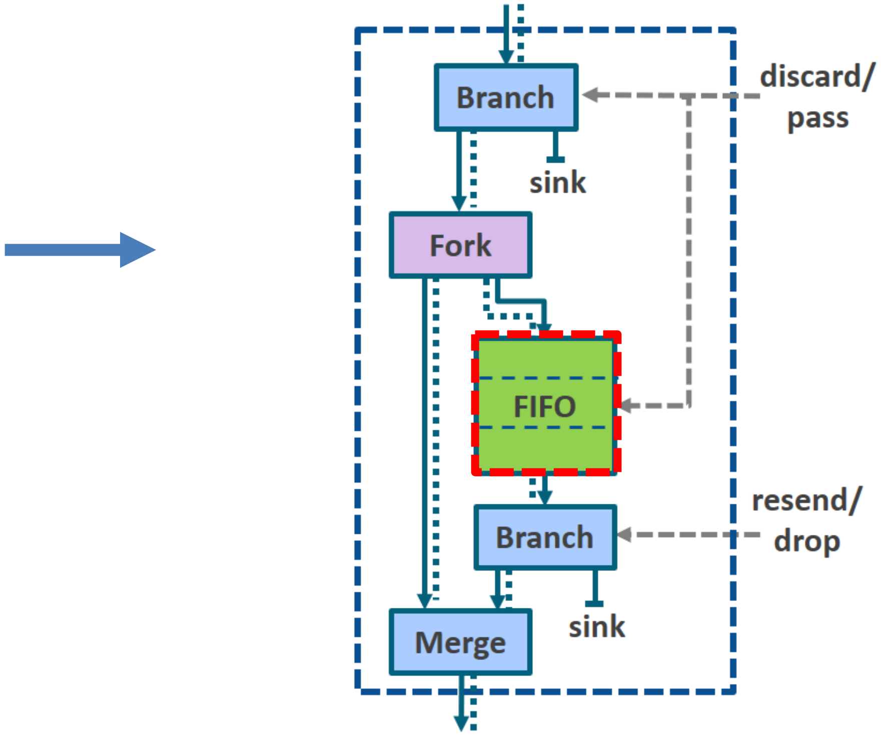
**Save unit:**

- Saves and reissues tokens



**Save-Commit unit:**

- Lets speculative tokens pass
- Discards misspeculated tokens
- Saves and reissues tokens

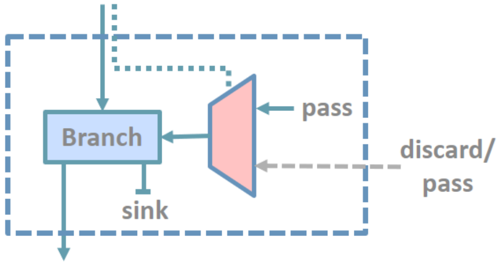


# Increasing Performance

- Merging the Save and Commit unit on cyclic paths

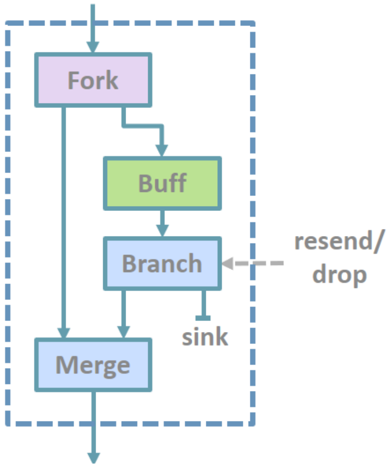
**Commit unit:**

- Stalls speculative tokens
- Discards misspeculated tokens



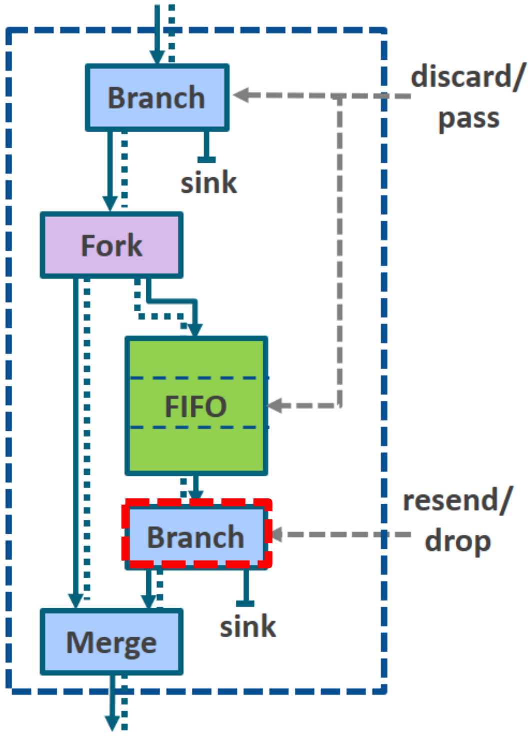
**Save unit:**

- Saves and reissues tokens



**Save-Commit unit:**

- Lets speculative tokens pass
- Discards misspeculated tokens
- Saves and reissues tokens

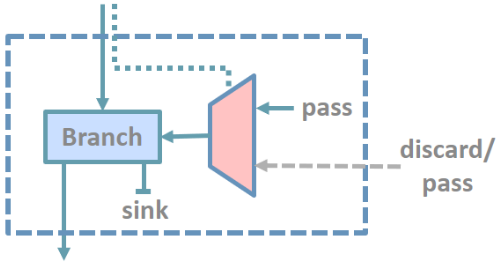


# Increasing Performance

- Merging the Save and Commit unit on cyclic paths

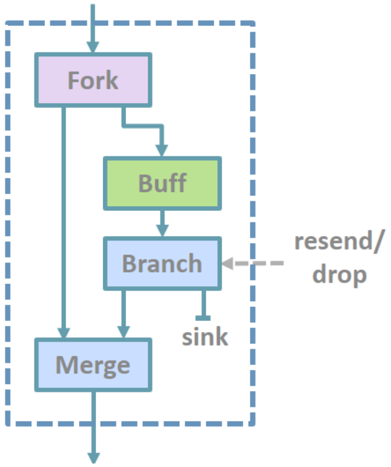
**Commit unit:**

- Stalls speculative tokens
- Discards misspeculated tokens



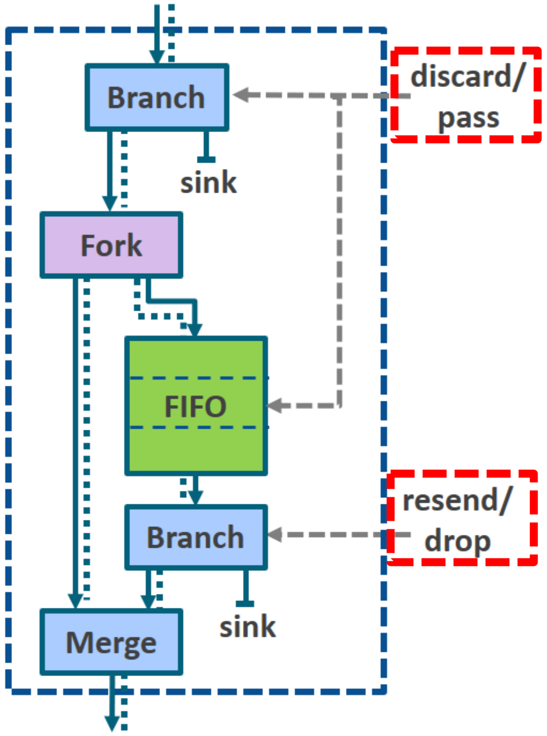
**Save unit:**

- Saves and reissues tokens



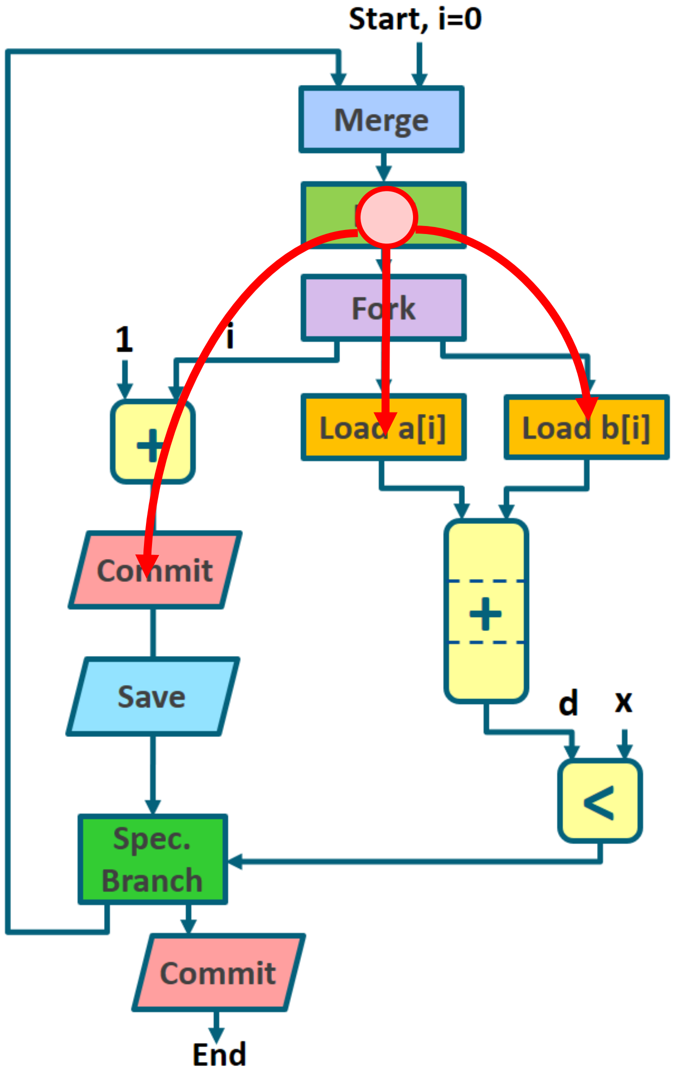
**Save-Commit unit:**

- Lets speculative tokens pass
- Discards misspeculated tokens
- Saves and reissues tokens



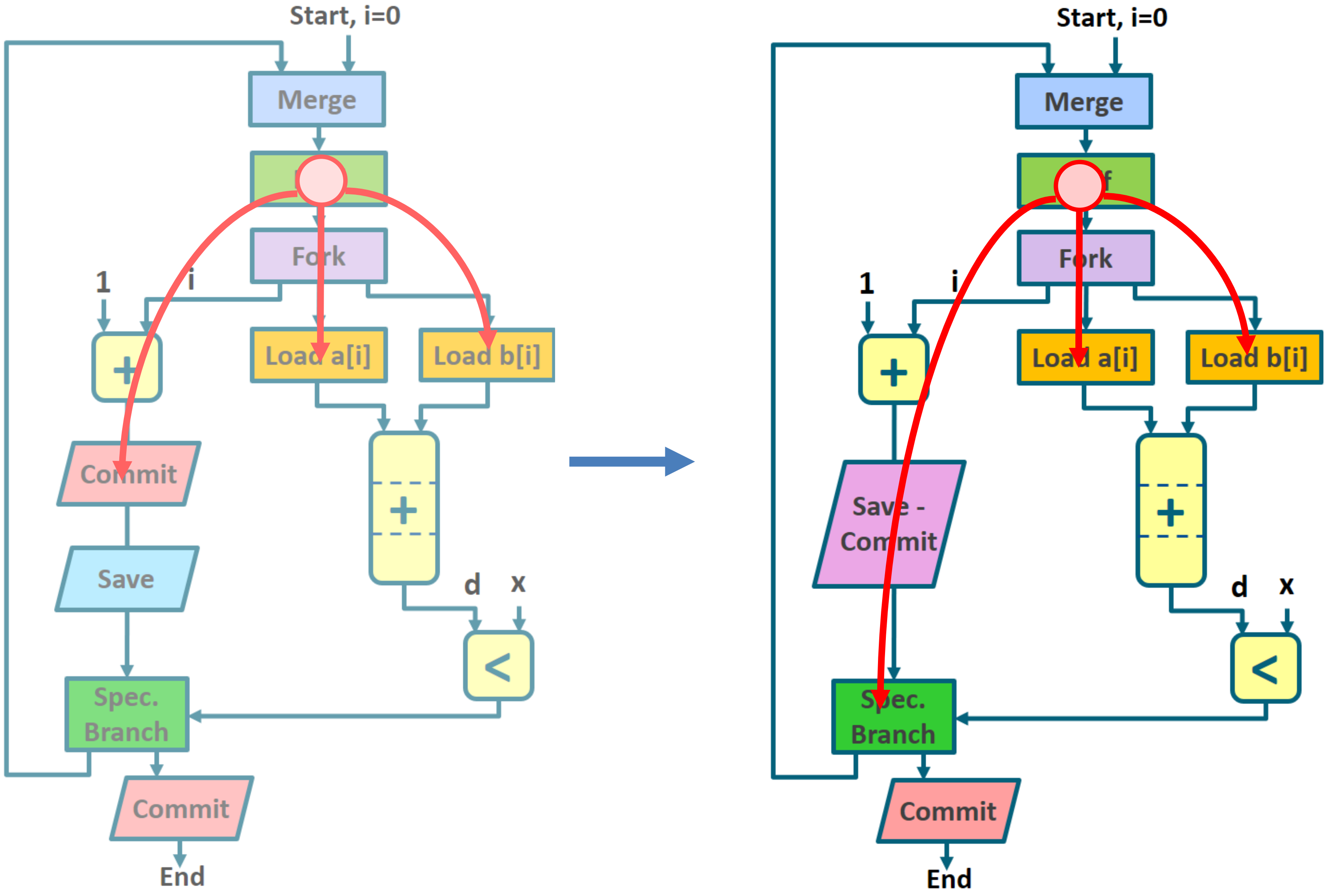
# Increasing Performance

- Merging the Save and Commit unit on cyclic paths

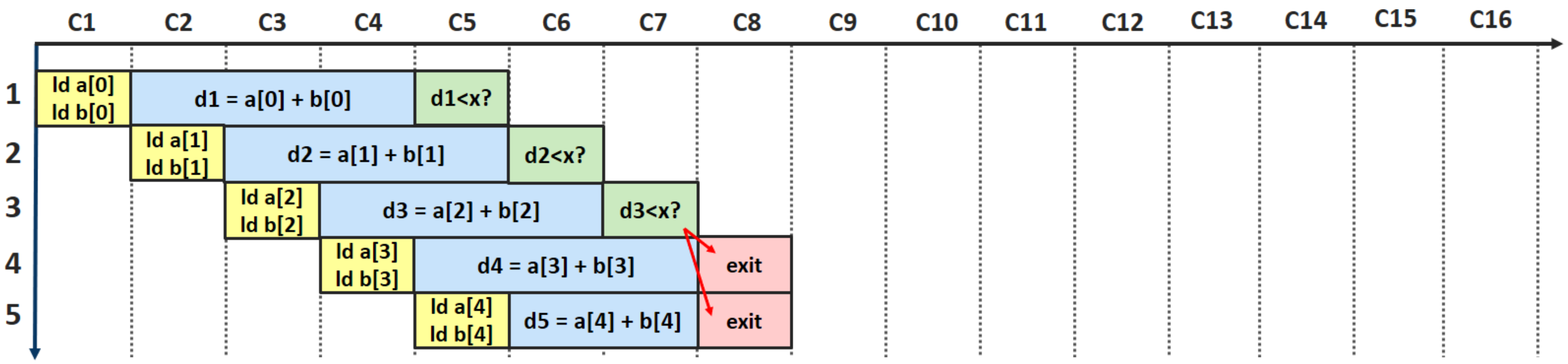


# Increasing Performance

- Merging the Save and Commit unit on cyclic paths



# Increasing Performance



C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15 C16

1 ld a[0] ld b[0] | d1 = a[0] + b[0] | d1<x?

2 ld a[1] ld b[1] | d2 = a[1] + b[1] | d2<x?

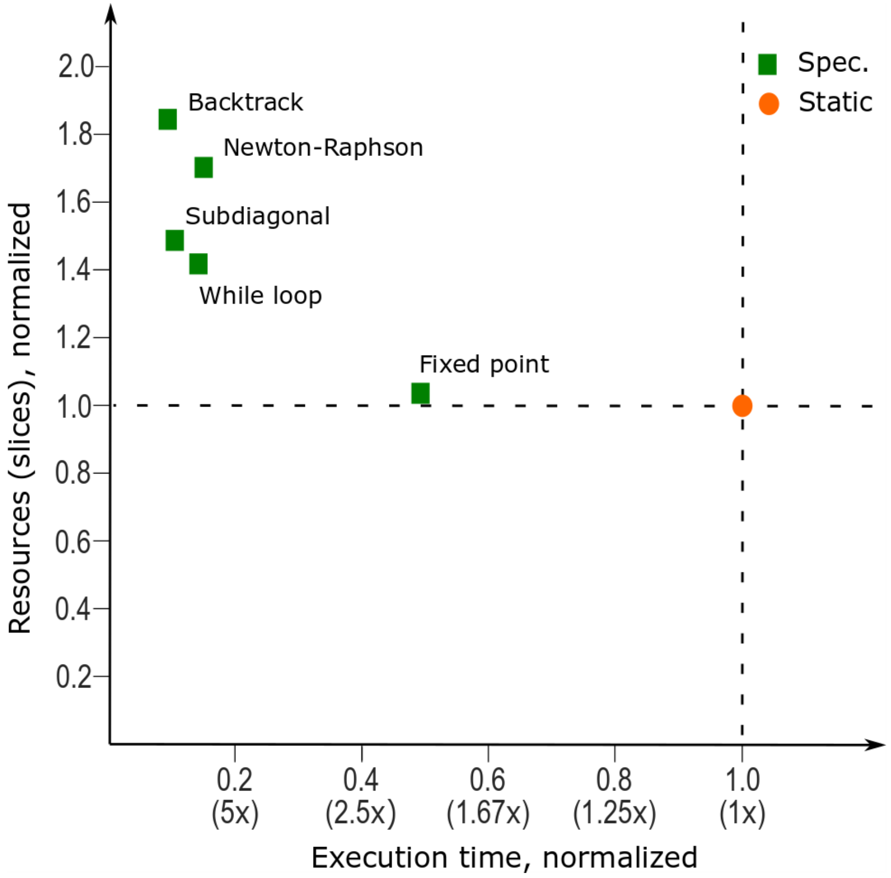
3 ld a[2] ld b[2] | d3 = a[2] + b[2] | d3<x?

4 ld a[3] ld b[3] | d4 = a[3] + b[3] | exit

5 ld a[4] ld b[4] | d5 = a[4] + b[4] | exit

# Results

- Timing and resources: traditional HLS (Static) and dataflow circuits with speculation (Speculative)
  - – Cases where dynamic scheduling on its own cannot achieve high parallelism [1]



[1] Press et al. Numerical Recipes: The Art of Scientific Computing. Cambridge University Press, 3rd edition.

# Part 3 Outline

✓ What traditional HLS does not do well

✓ Synthesis of dataflow circuits

✓ Buffers and performance

✓ The problem with memory

✓ Conquering new grounds with speculation

# What to Expect from Dynamic HLS?

- Two **hopes** derived from the VLIW vs. OoO analogy:
  - Significantly better performance in control dominated applications with poorly predictable memory accesses
  - Better out-of-the-box performance
- The former is almost certain, the second less so
- A major issue is the **hardware overhead** of supporting dynamic schedules
  - Probably tolerable for the bulk of the circuits
  - Yet, LSQs represent quite tangible overheads, esp. in FPGAs (but could be hardened there)
- Probably statically-scheduled HLS remains the best choice for classic DSP-like applications

# Conclusions
# on Processor Customization and HLS

- Customizable processors and high-level synthesis are promising techniques to **accelerate program execution through customized hardware**
- We may need more of this in a post-Moore (no transistor scaling) and post-Dennard (no power scaling) world
- Yet, all these techniques require **considerable manual work and expertise** (at least to obtain decent results)
- Current research tries both to improve the quality of HLS-generated circuits as well as moving further up the level of abstraction to extract more easily more parallelism (e.g., domain-specific languages)

# References

**Part 1**

- J. A. Fisher, *Customized Instruction-Sets for Embedded Processors*, DAC, June 1999
- P. Ienne and R. Leupers, eds., *Customizable Embedded Processors*, MK, 2006

**Part 2**

- G. Martin and G. Smith, *High-Level Synthesis: Past, Present, and Future*, IEEE Design & Test of Computers, July/August 2009
- J. Cong and Z. Zhang, *An efficient and versatile scheduling algorithm based on SDC formulation*, DAC, July 2006
- R. Kastner, J. Matai, and S. Neuendorffer, *Parallel Programming for FPGAs*, https://arxiv.org/abs/1805.03648, May 2018

**Part 3**

- L. Josipović, R. Ghosal, and P. Ienne, *Dynamically Scheduled High-level Synthesis*, ISFPGA, February 2018
- L. Josipović, Ph. Brisk, and P. Ienne, *An Out-of-Order Load-Store Queue for Spatial Computing*, ACM TECS, September 2017
- L. Josipović, A. Guerrieri, and P. Ienne, *Speculative Dataflow Circuits*, ISFPGA, February 2019
- L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, Jordi Cortadella, *Buffer Placement and Sizing for High-Performance Dataflow Circuits*, ISFPGA, February 2020 (Best Paper Award)
- L. Josipović, A. Guerrieri, P. Ienne, *Dynamatic: From C/C++ to Dynamically Scheduled Circuits*, ISFPGA, February 2020

**dynamatic.epfl.ch**