High-Level Synthesis

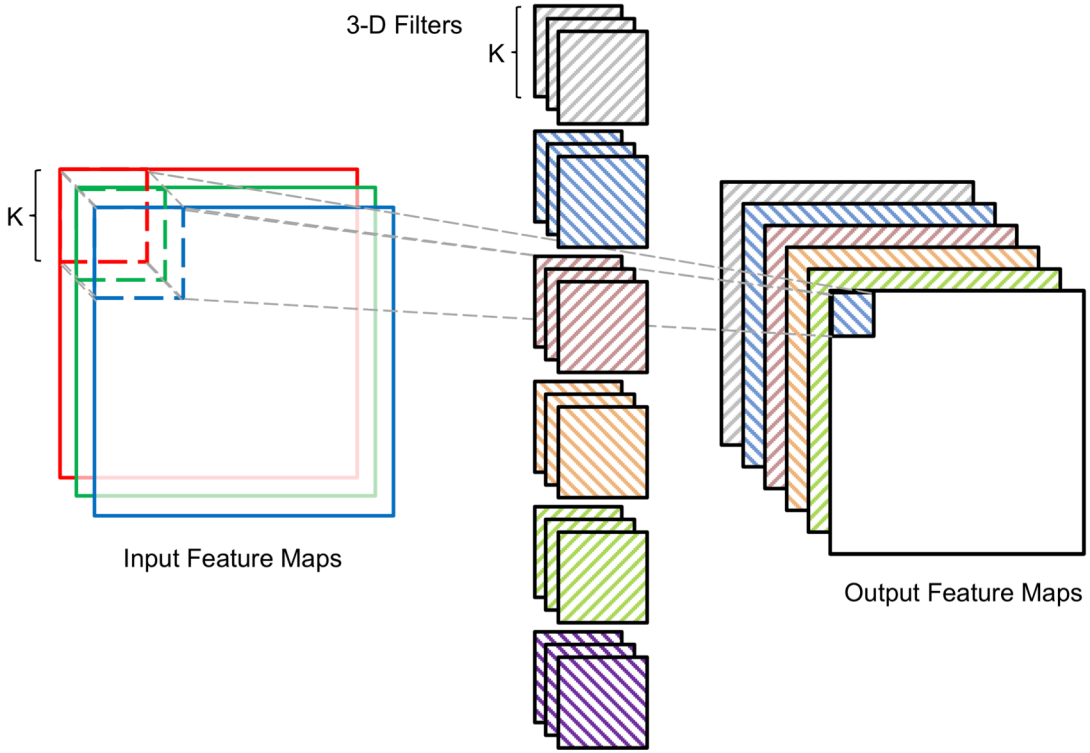
Jason Anderson

janders@eecg.toronto.edu

Goals for Today

- High-level synthesis:
  - What is it?
  - Main steps of HLS
  - Overview of underlying HLS algorithms
  - Limitations of HLS
- Ways to improve auto-generated HW:
  - Constraints to the HLS tool
  - Change your input program

Lab Portion



- Use HLS to implement convolution in hardware, as in a convolutional neural network (CNN)
- Use HLS constraints + code changes to achieve nearly 5X performance improvement vs. baseline

Introduction – Who Am I?

- Professor in the Computer Engineering section of the ECE department of the University of Toronto.
- Joined the university in 2008 after working about 10 years in the computer chip (semiconductor) industry…

- In my career, I went back and forth between the university and industry…


Winnipeg, Canada


CANADA
ARCTIC OCEAN
Ellesmere Island
GREENLAND
0 600 km
0 360 miles
ALASKA (USA)
Melville Island
Banks Island
Devon Island
Baffin Bay
Arctic Circle
Victoria Island
Baffin Island
Dawson City
Nunavut
Auyuittua
Yukon
Northwest Territories
ATLANTIC OCEAN
Whitehorse
Watson Lake
Yellowknife
Great Slave Lake
Hudson Bay
Newfoundland
British Columbia
Alberta
Wood Buffalo
Churchill
L'Anse-aux-Meadows
Prince George
Manitoba
St John's
Rocky Mountains
Edmonton
Quebec
Prince Edward Is
Vancouver
Saskatchewan
Lake Winnipeg
Gaspésie Park
Charlottetown
The Prairies
Victoria
Calgary
Ontario
Bay of Fundy
Regina
Winnipeg
Quebec City
Halifax
Lunenburg
Montreal
Algonquin Park
Ottawa
UNITED STATES OF AMERICA
Toronto
Niagara Falls
ATLANTIC OCEAN


Manitoba, Canada


Winnipeg, Canada

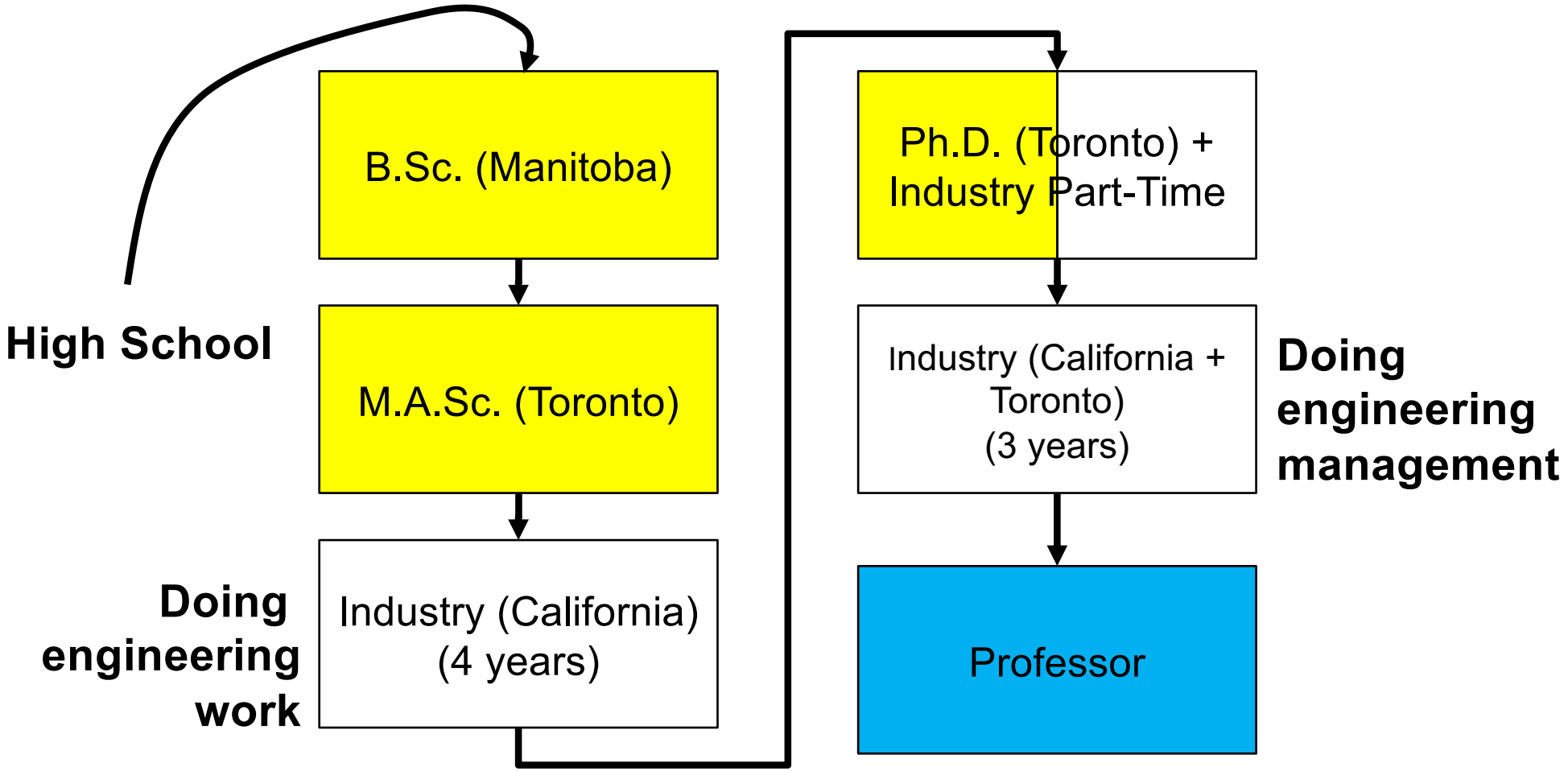

University of Manitoba
THE UNIVERSITY OF MANITOBA
NEW


University of Toronto

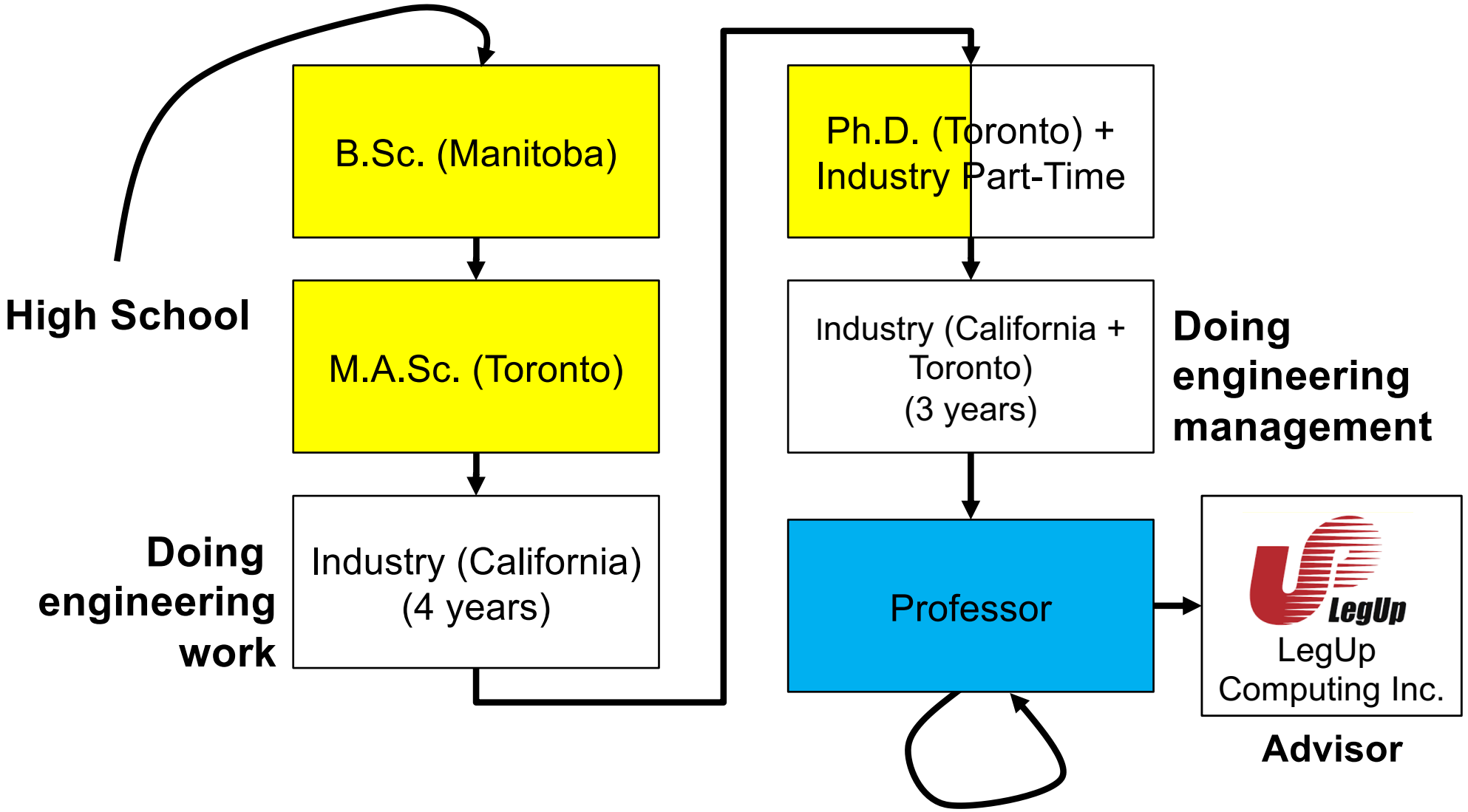

Toronto, Canada


SILICON VALLEY
San Francisco
Oakland
Berkeley
Fremont
Pleasanton
Dublin
San Ramon
Hayward
South San Francisco
San Mateo
Redwood City
Stanford University
Palo Alto
Mountain View
Sunnyvale
Santa Clara
Milpitas
Campbell
Saratoga
Cupertino
XILINX

My Career Path



My Career Path



Spawn of LegUp Computing Inc.

- Three key student contributors interested in commercialization

Andrew Canis,
Ph.D, CEO

Jongsok Choi,
Ph.D, CTO

Ruolong Lian,
M.A.Sc, COO

Prof. Jason Anderson
Chief Scientific Advisor

- Incorporated in 2015
- Seed funding from Intel in 2018

Microchip Acquires High-Level Synthesis Tool Provider LegUp to Simplify Development of PolarFire FPGA-based Edge Compute Solutions

Software engineers can now map applications coded in C/C++ directly into PolarFire FPGAs and SoCs that are the industry's lowest-power mid-range fabric solutions for acceleration

CHANDLER, Ariz., Oct. 21, 2020 — Microchip Technology Inc. **(Nasdaq: MCHP)** today announced it acquired Toronto-based LegUp Computing Inc., expanding its Field-Programmable Gate Array (FPGA)-based edge compute solution stack with a high-level synthesis (HLS) tool. Commercialized from University of Toronto research, the LegUp HLS tool will make it easier for a larger community of software engineers to harness the algorithm-accelerating power of Microchip's PolarFire® FPGA and PolarFire System on Chip (SoC) platforms.

Computations in Two Ways

Computations in Two Ways

Write Software

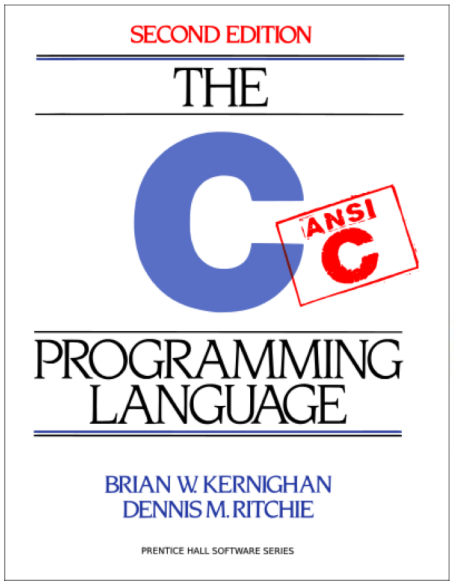
Computations in Two Ways

Write Software

Computations in Two Ways

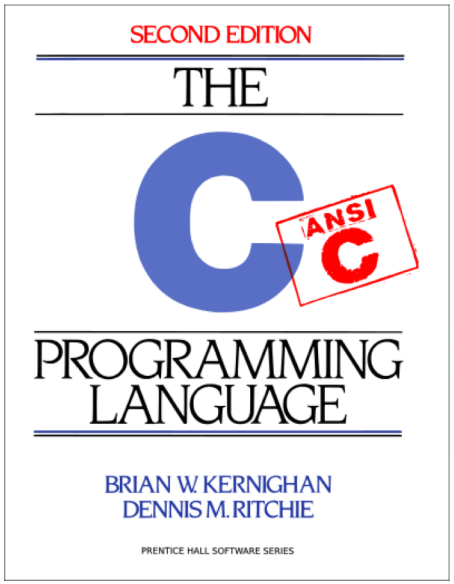
Write Software

Computations in Two Ways



Write Software

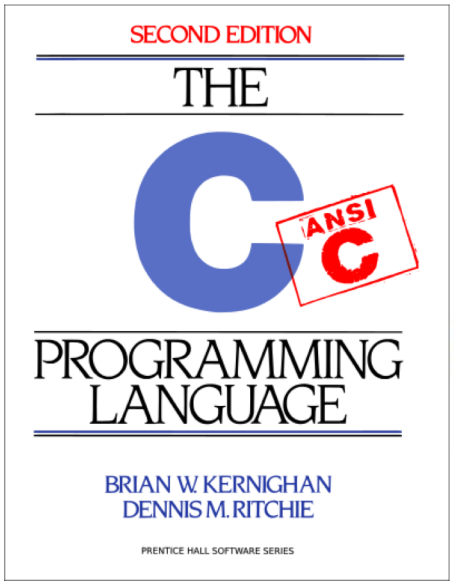
Computations in Two Ways



Write Software

Design Custom Circuits

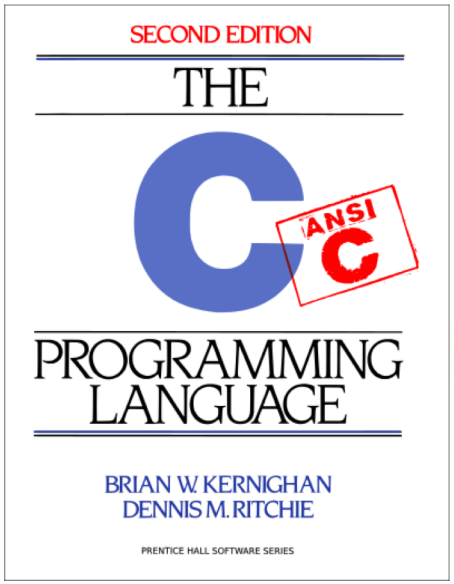
Computations in Two Ways



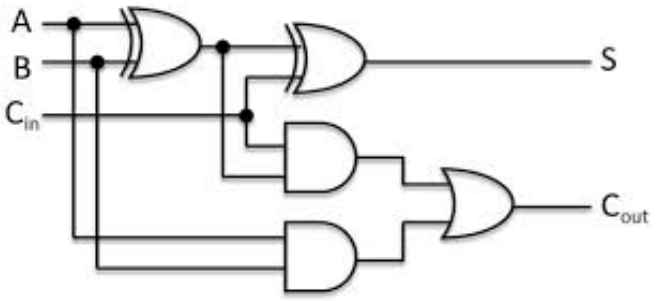
Write Software

Design Custom Circuits

Computations in Two Ways



Write Software



Design Custom Circuits

Design Methodology

Design Methodology

Write software

Design Methodology

Write software

- Easy

Design Methodology

Write software

- Easy
- Flexibility → lower performance

Design Methodology

Write software

- Easy
- Flexibility → lower performance

Design Custom Circuits

Design Methodology

Write software

- Easy
- Flexibility → lower performance

Design Custom Circuits

- Efficient, low power

Design Methodology

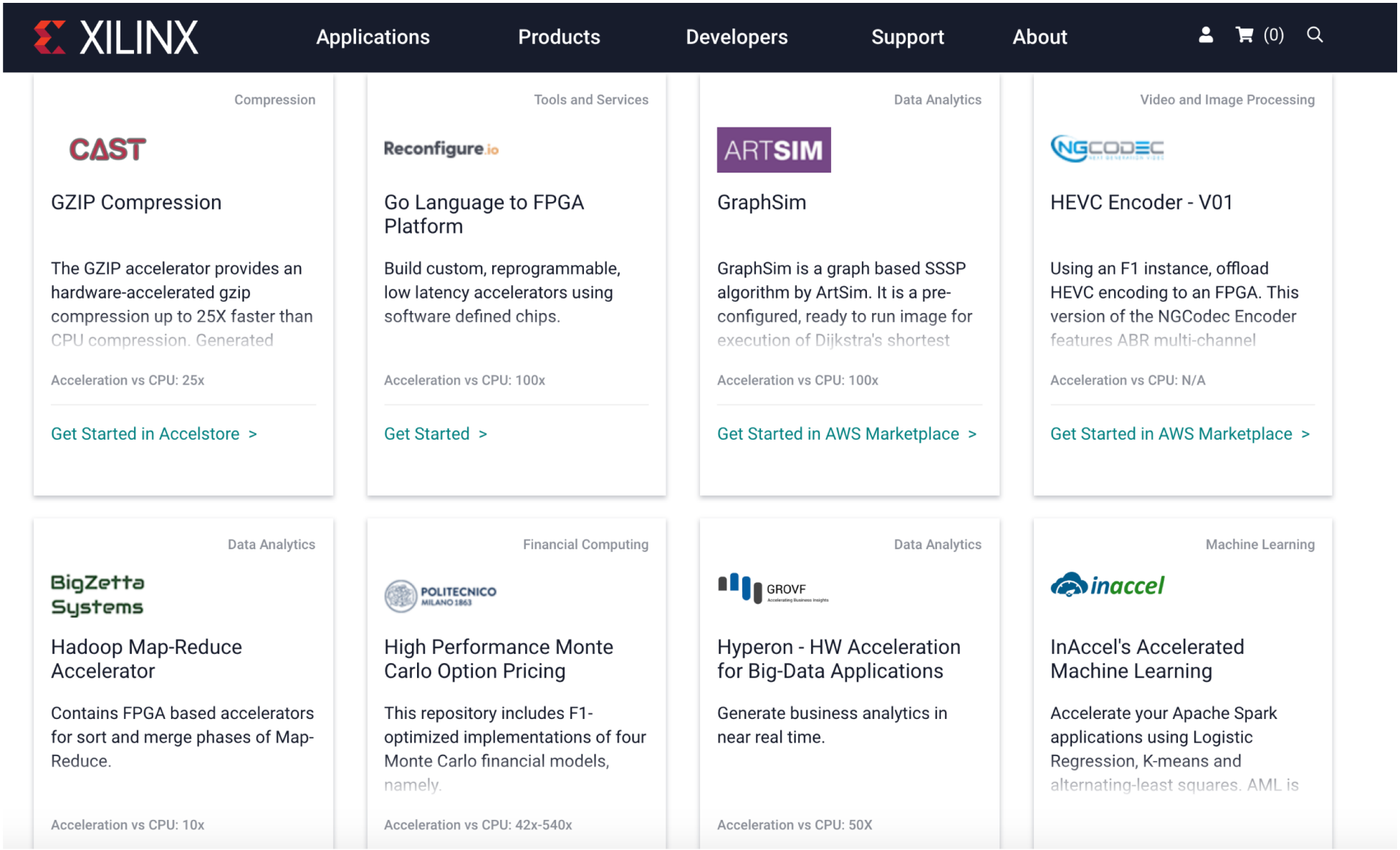
Write software

- Easy
- Flexibility → lower performance

Design Custom Circuits

- Efficient, low power
- Need specialized knowledge

FPGA Hardware's Potential



Hardware Design is Difficult

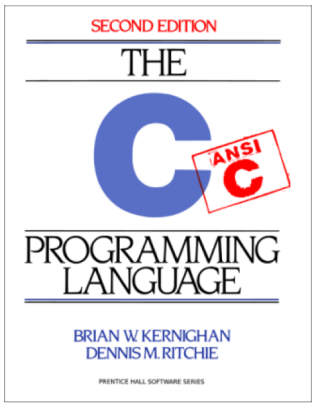
- FPGA "success stories" are pervasive, yet the technology remains inaccessible to software engineers
  - Requires use of hardware description languages: Verilog and VHDL
- Hardware design skills are rare:
  - 10 software engineers for every hardware engineer*

*US Bureau of Labor Statistics

Hardware Design is Difficult

- FPGA "success stories" are pervasive, yet the technology remains inaccessible to software engineers
  - Requires use of hardware description languages: Verilog and VHDL
- Hardware design skills are rare:
  - 10 software engineers for every hardware engineer*
- What is needed:
  1. Make hardware design easier for hardware engineers
  2. Allow software engineers to design hardware and reap its energy/performance benefits

A Solution

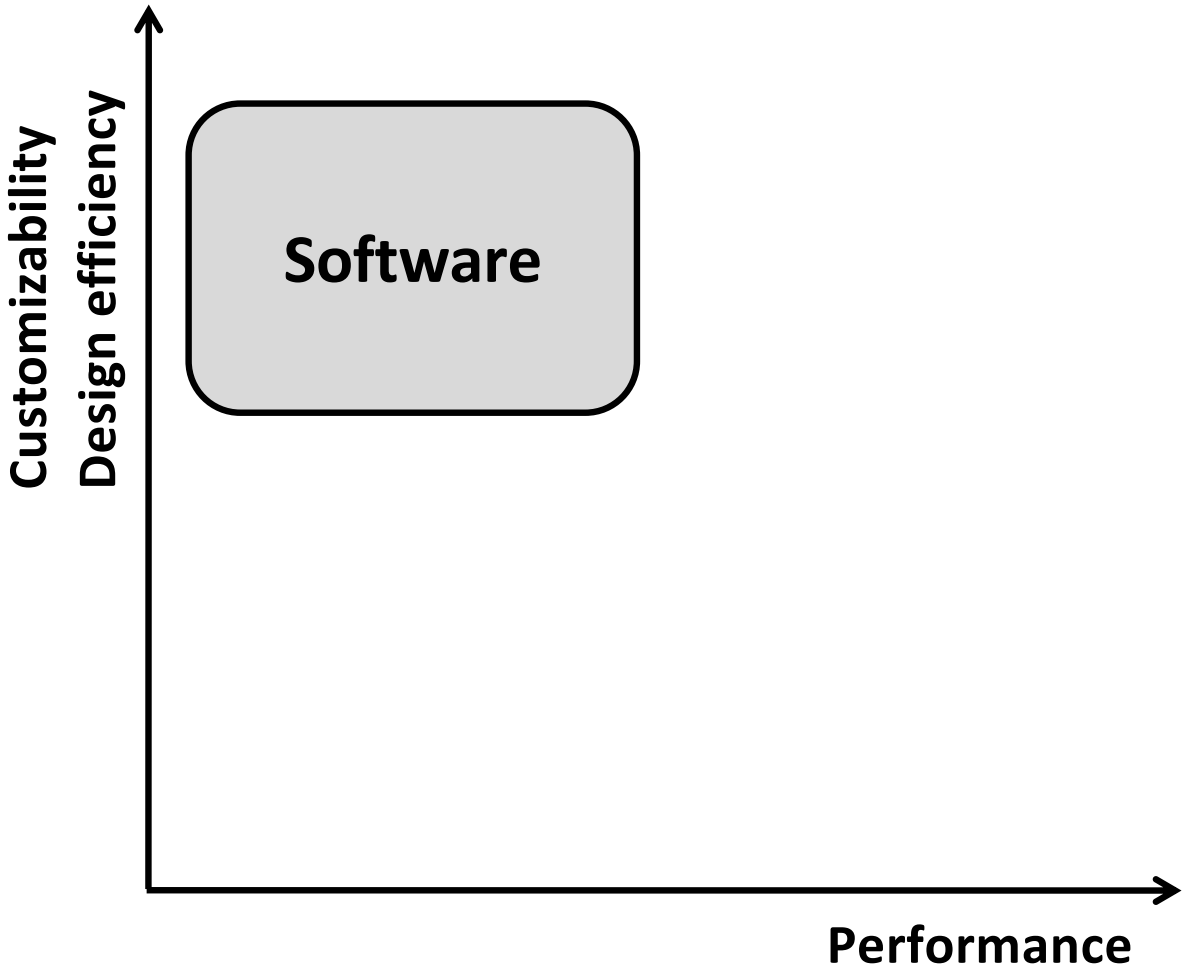


Flexibility/
Ease of Use

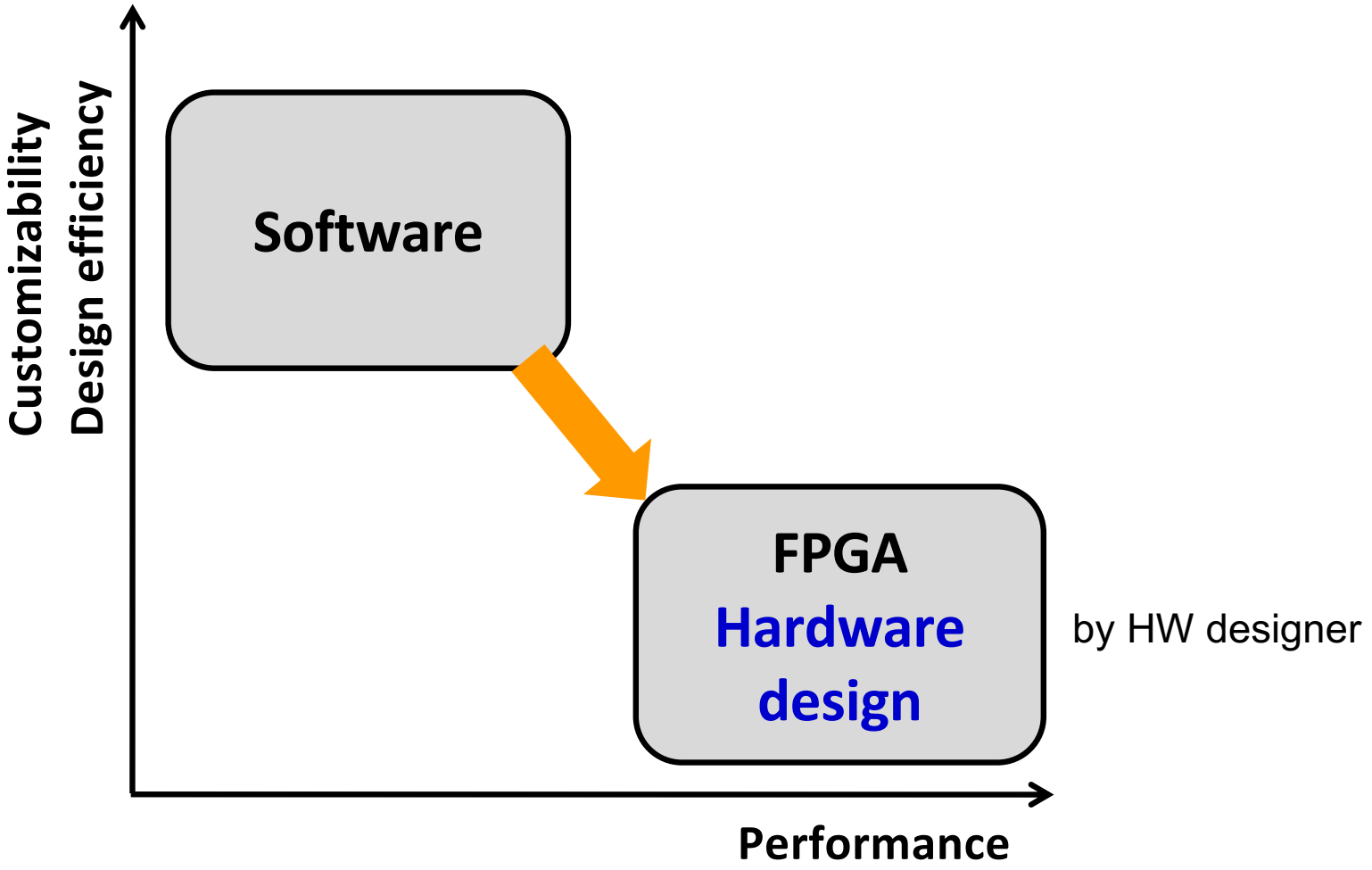
High-level
Sythesis

High-performance/
Energy-efficiency

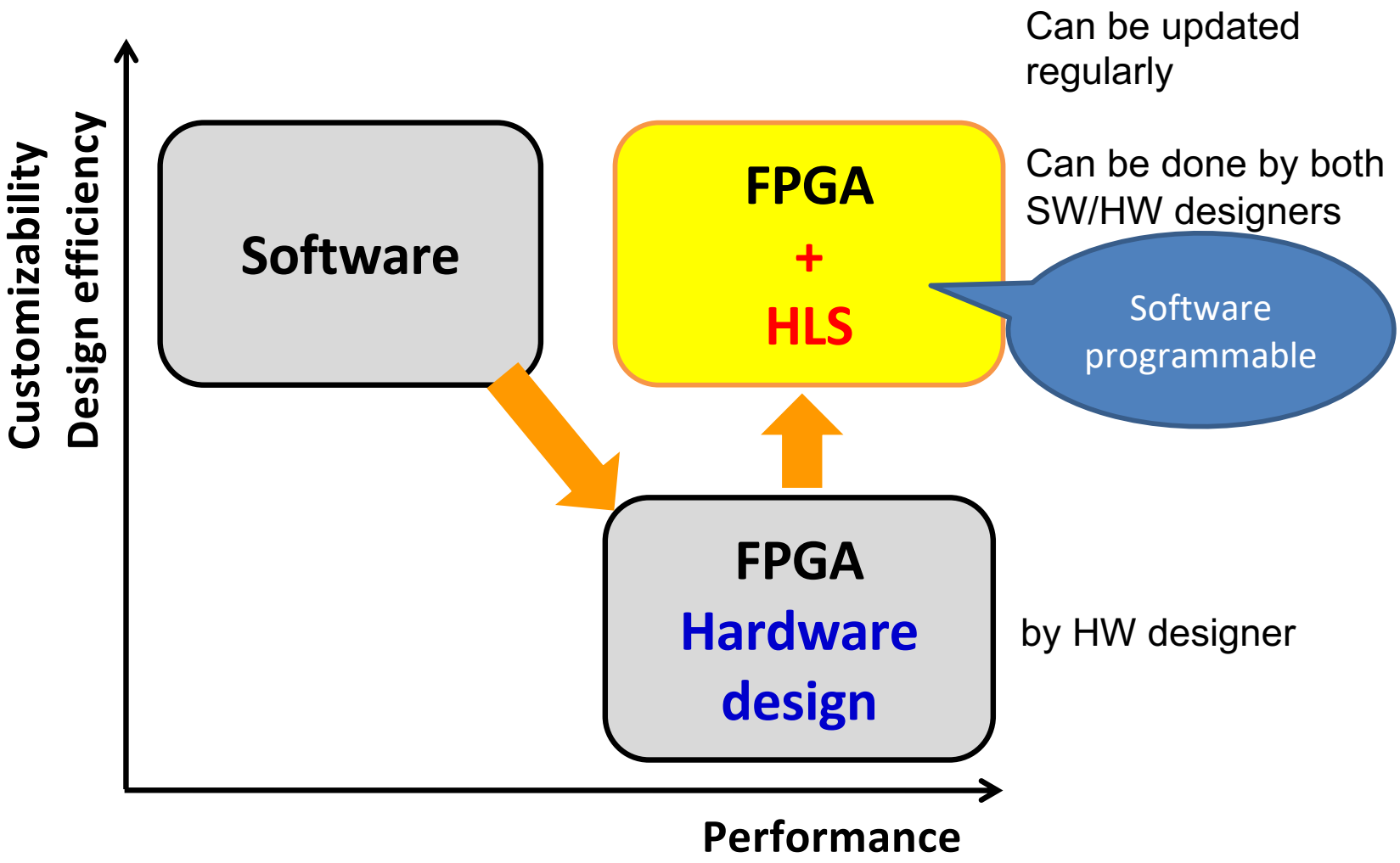
HLS Value Proposition



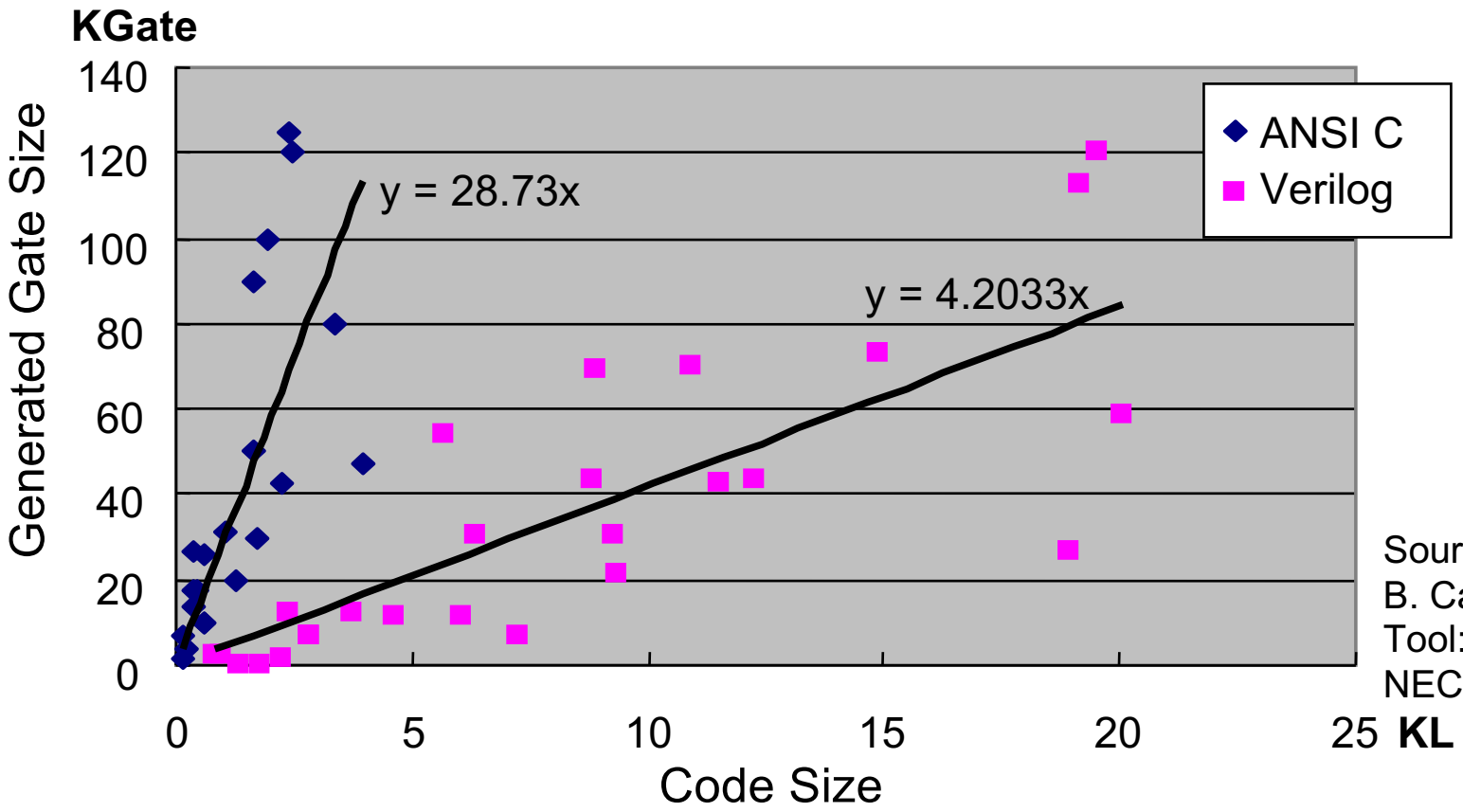
HLS Value Proposition



HLS Value Proposition



Productivity



Source:
B. Carrion Schafer
Tool:
NEC CWB

RTL (Verilog) 4 gates / line

Behavioral C 28 gates / line

➔ **7x more productive**

Benefits of HLS

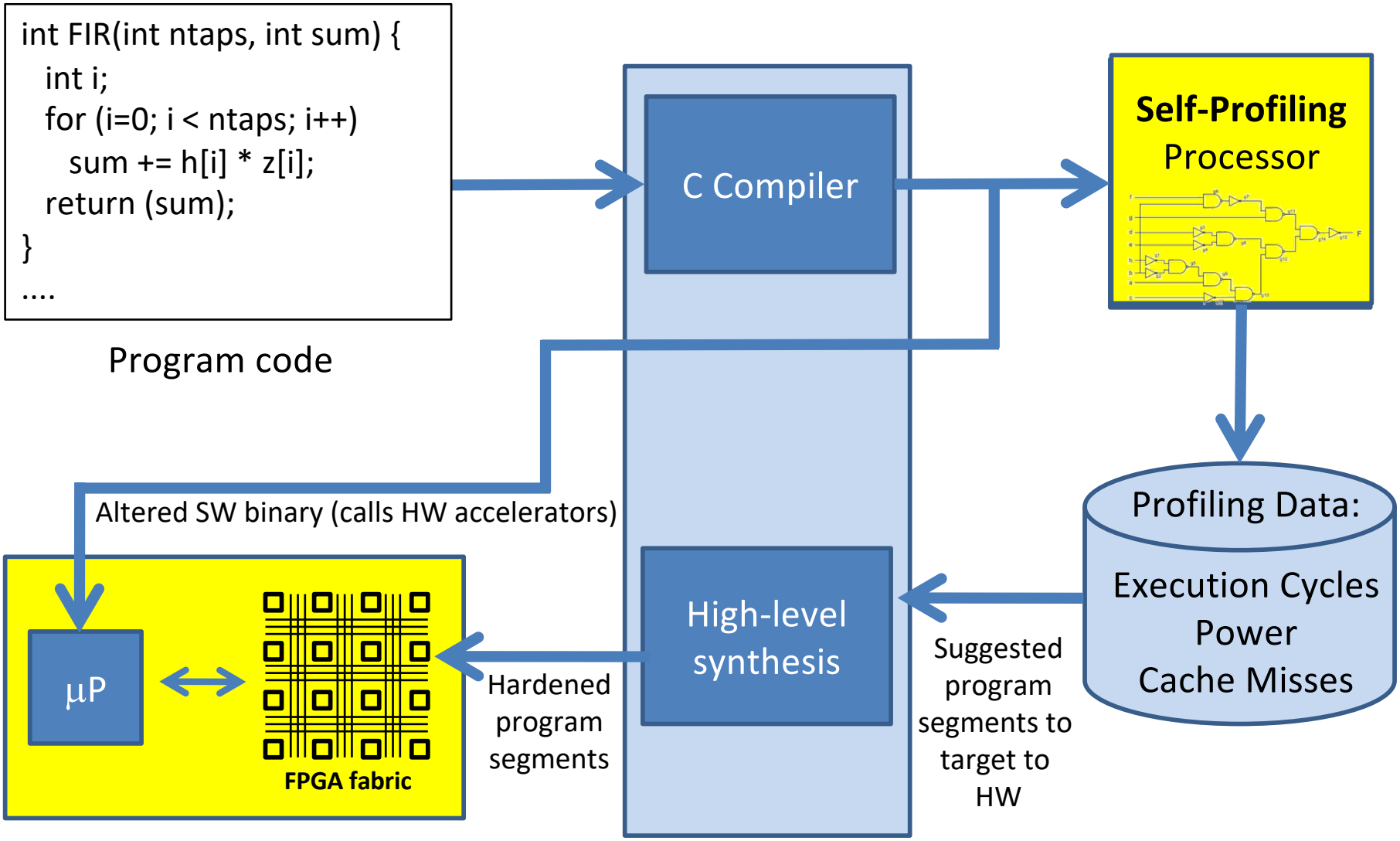
- Shorter time-to-market (lower NRE)
- Easier modifiability/maintainability
  - Design spec is in SW
  - Important for some appls where spec isn't firm or changes frequently, e.g. finance models
- Rapid exploration of HW solution space
- Make FPGA HW accessible to SW engineers
  - Bring the energy and speed benefits of HW to those with SW skills

LegUp HLS for FPGAs

- HLS framework that takes a C program as input, and compiles to either:
  - 1) hardware alone
  - 2) a hybrid processor/accelerator system
- Under development since 2009
- 5000+ downloads since first release in 2011
  - License for non-commercial research purposes

http://legup.eecg.toronto.edu

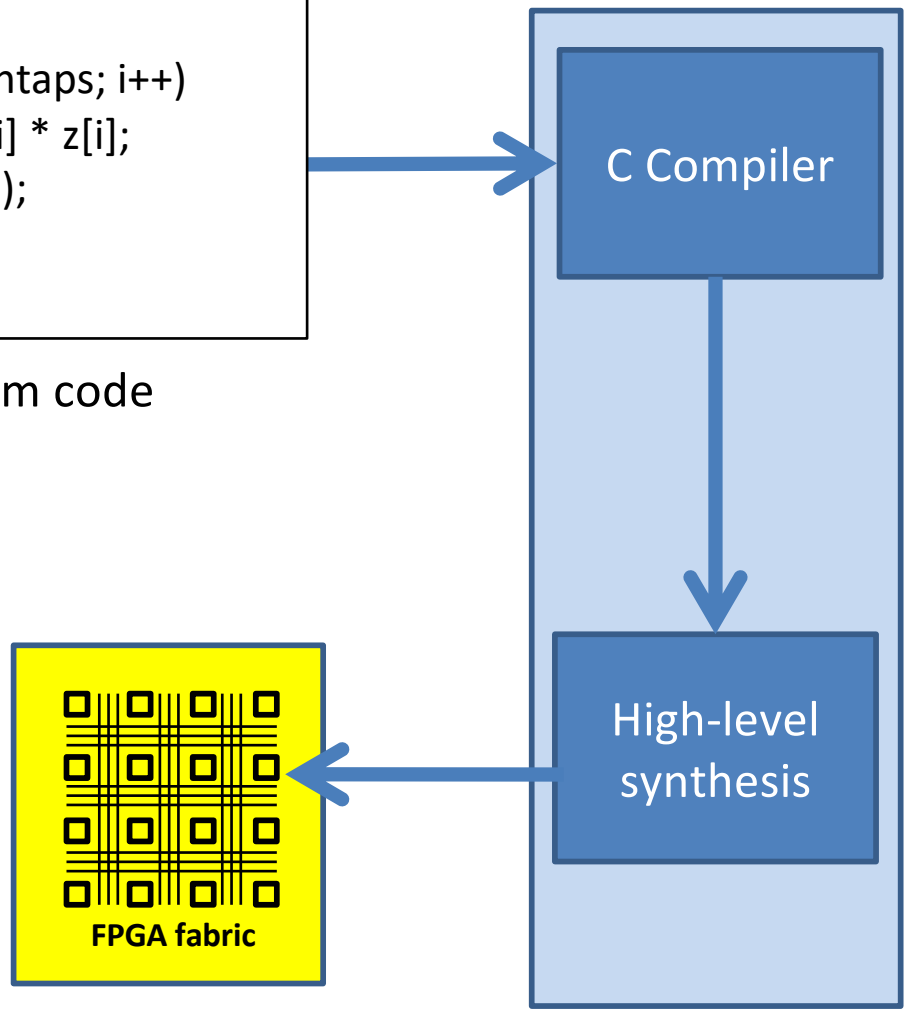
LegUp Processor/Accelerator Flow



LegUp Pure HW Flow

```
int FIR(int ntaps, int sum) {
  int i;
  for (i=0; i < ntaps; i++)
    sum += h[i] * z[i];
  return (sum);
}
....
```

Program code



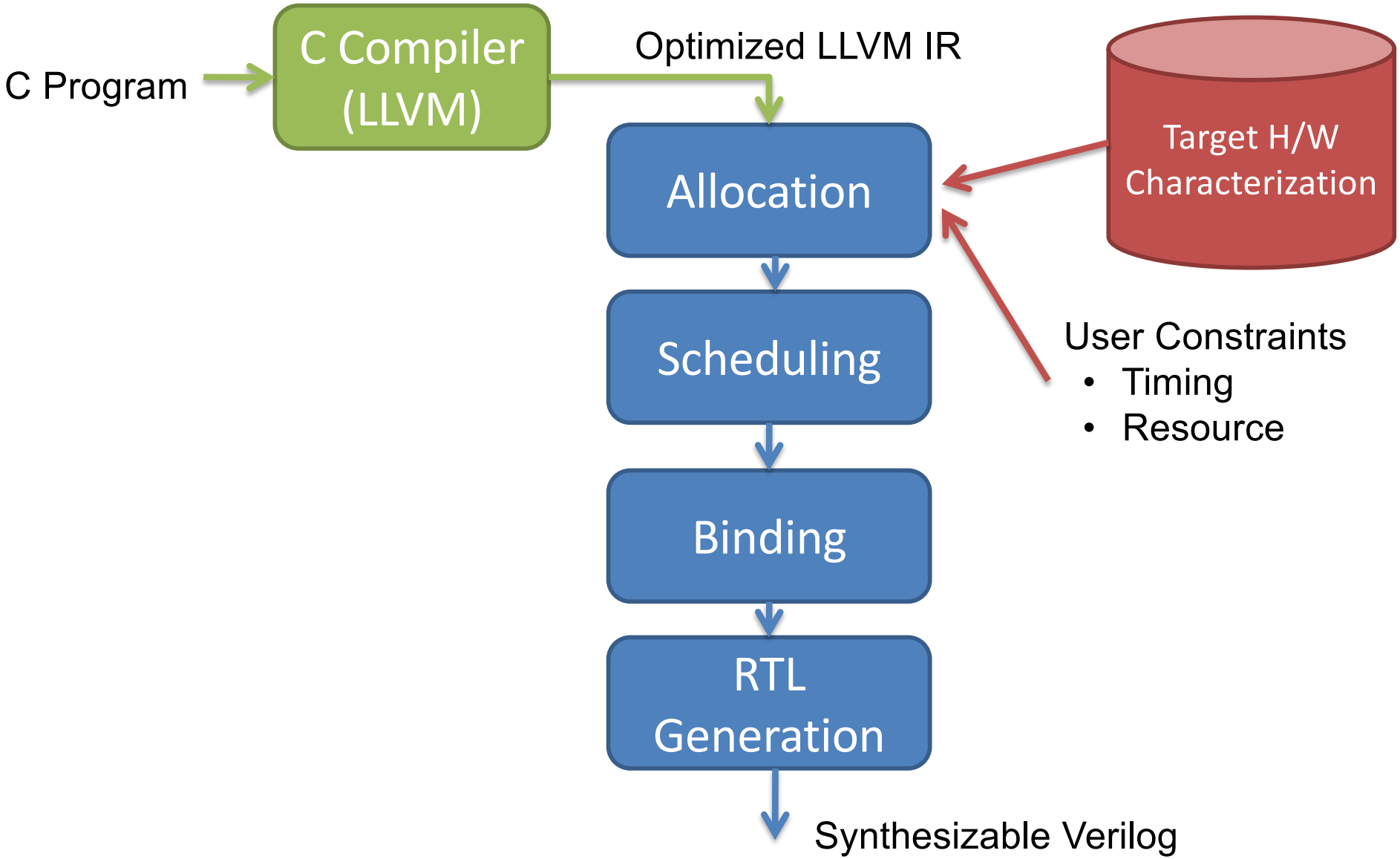
High-Level Synthesis Framework

- Leverage LLVM compiler infrastructure:
  - Language support: C
  - Standard compiler optimizations
- We support a large subset of ANSI C:

Supported	Unsupported
Functions	Dynamic Memory (FPL'19)
Arrays, Structs	Recursion
Global Variables	
Pointer Arithmetic	
Floating Point	

How Does High-Level Synthesis Work?

High-Level Synthesis Flow



LLVM Compiler

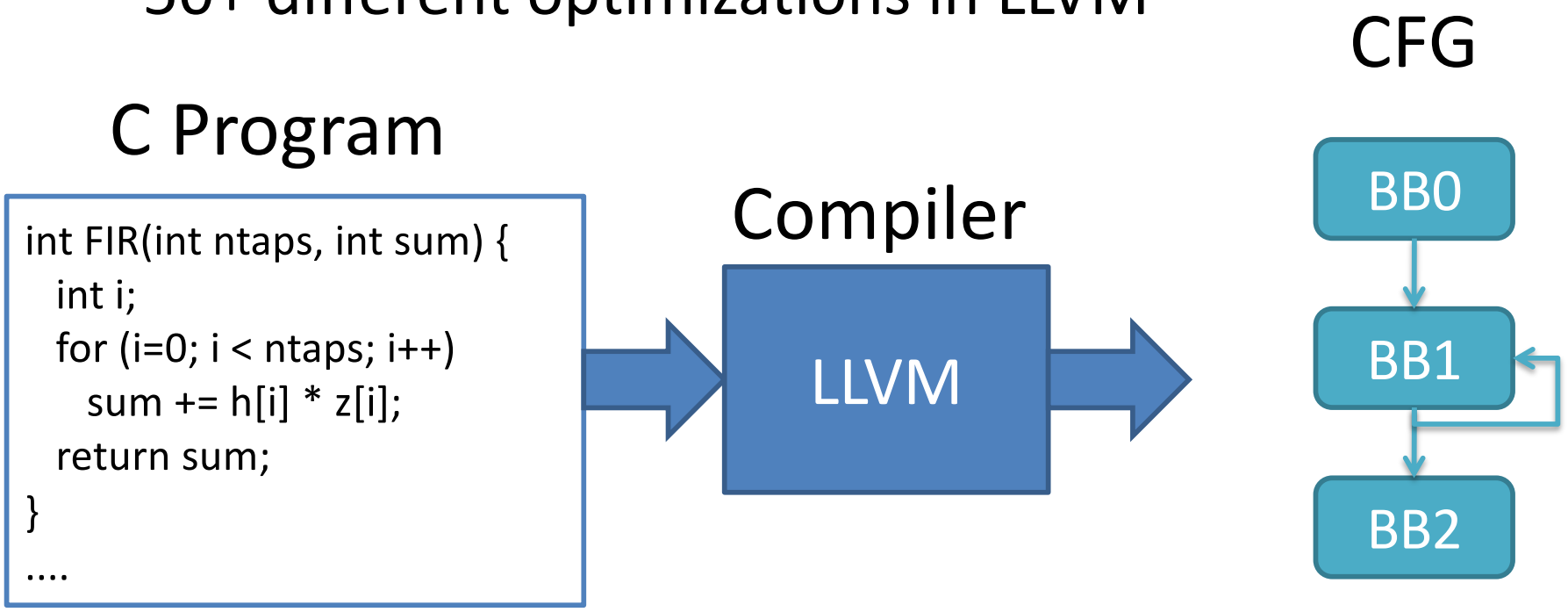
LLVM Compiler

- Open-source compiler framework
  - http://llvm.org
- Used by Apple, NVIDIA, AMD, Xilinx, others
- Competitive quality with gcc
- LegUp HLS is a “back-end” of LLVM

- LLVM: low-level virtual machine

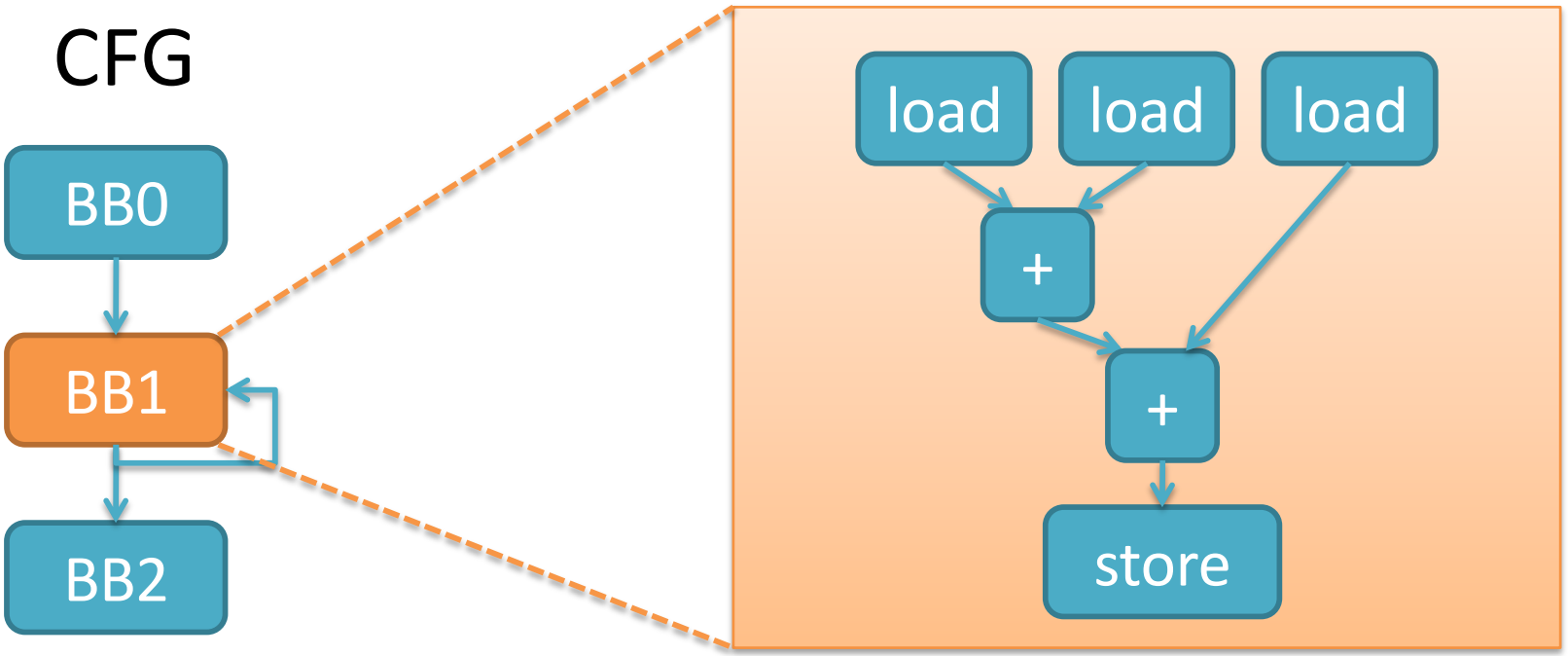
LLVM Compiler

- LLVM will compile C code into a **control flow graph (CFG)**
- LLVM will perform standard optimizations
  - 50+ different optimizations in LLVM



Control Flow Graph

- Control flow graph is composed of **basic blocks**
- **basic block**: is a sequence of instructions terminated with exactly one branch
  - Can be represented by an acyclic **data flow graph:**



LLVM Details

- Instructions in basic blocks are primitive computational operations:
  - shift, add, divide, xor, and, etc.
- Or are control-flow operations:
  - branch, call, etc.
- The CDFG is represented in LLVM's **intermediate representation** (IR)
  - LLVM IR is machine-independent assembly code

High-Level Synthesis: Scheduling

Scheduling: Key Aspect of HLS

- How to assign the computations of a program into the hardware time steps?
  - Defines the HW's finite state machine

C language snippet:

```
z = a+b;
x = c+d;
q = z+x;
q = q-2;
r = q*2;
```

Programs do not contain the notion of "time steps".

Here, we have:

3 add operations
1 subtract operation
1 multiplication operation

Scheduling

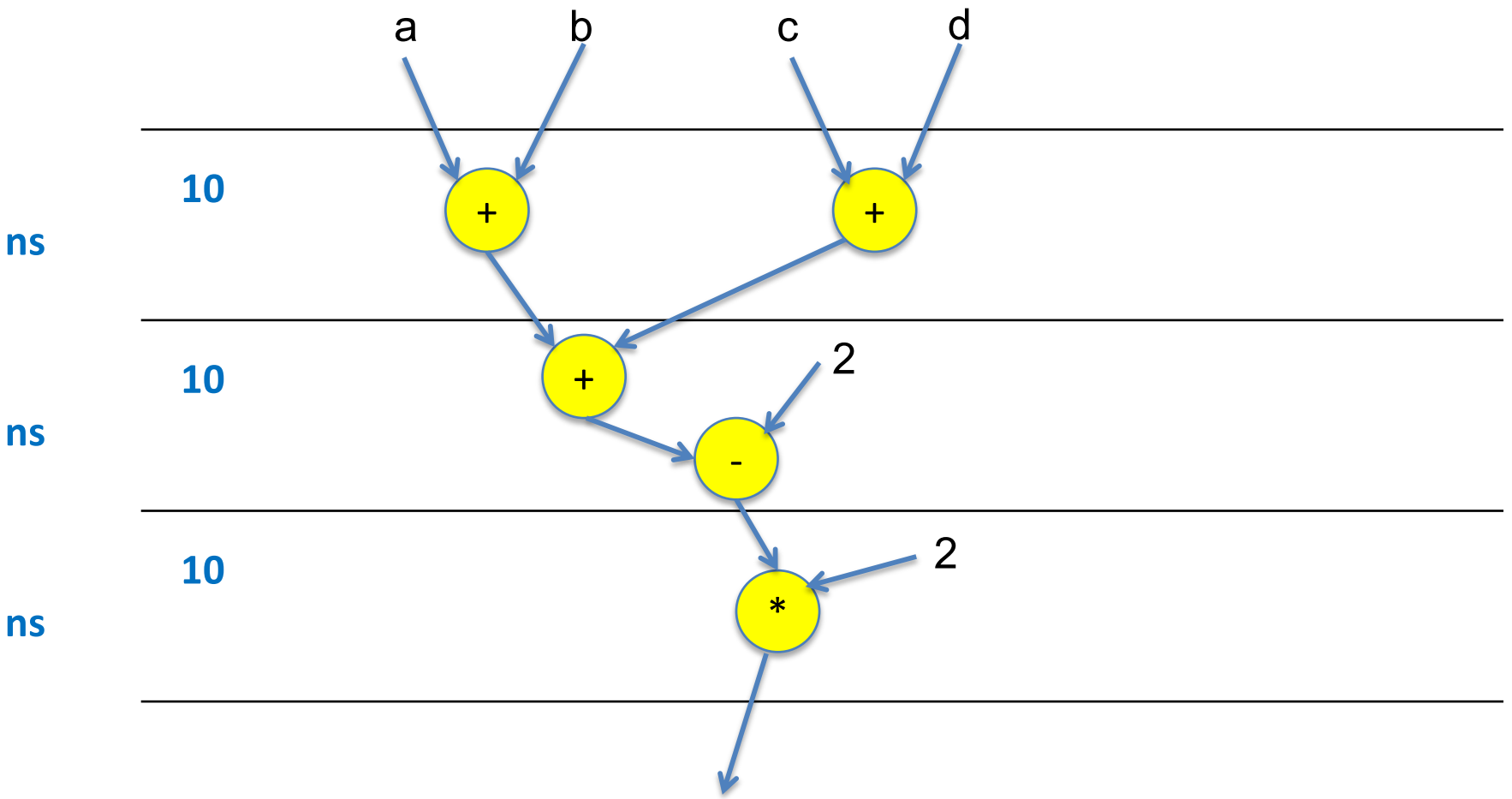
C language snippet:

```
z = a+b;
x = c+d;
q = z+x;
q = q-2;
r = q*2;
```

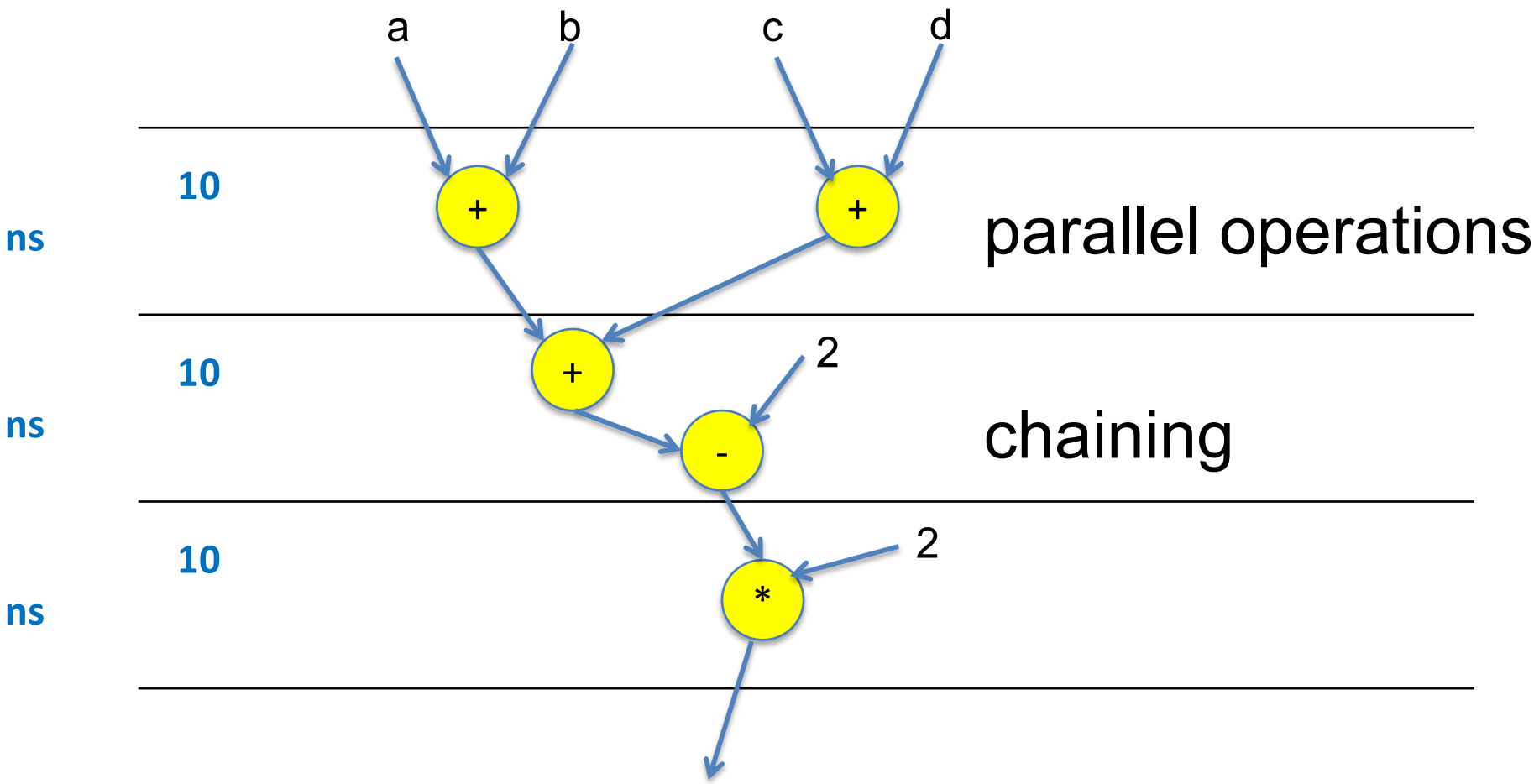
Questions:

- Which operations can be scheduled in the same time step?
- Which operations are dependent on others?
- If addition takes 5ns, subtraction takes 5ns and multiplication takes 10ns, how to schedule?
  - Target clock step length is 10ns

Scheduling



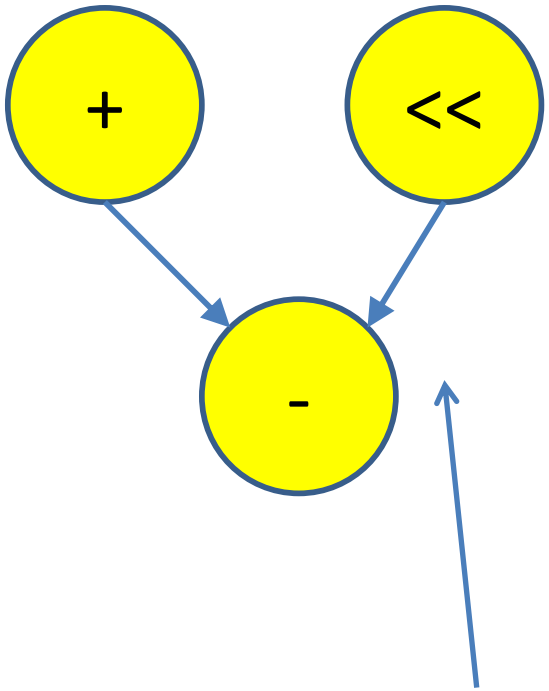
Scheduling



LegUp Uses "SDC Scheduling"

- SDC ↔ System of Difference Constraints
  - Cong, Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation". DAC 2006: 433-438.
- Basic idea: formulate scheduling as a mathematical optimization problem
  - Linear objective function + linear constraints (==, <=, >=)
- The problem is a linear program (LP)
  - Solvable in polynomial time with standard solvers

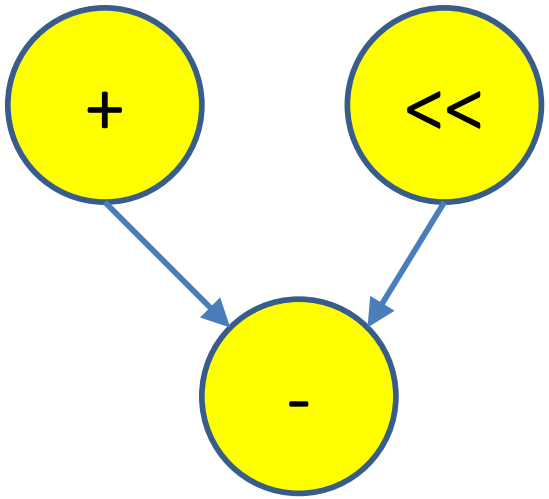
Define Variables



- For each operation *i* to schedule, create a variable x_i
- The x_i's will hold the cycle # in which each op is scheduled
- Here we have:
  - x_{add}, x_{shift}, x_{sub}

Data flow graph (DFG):
already accessible in LLVM

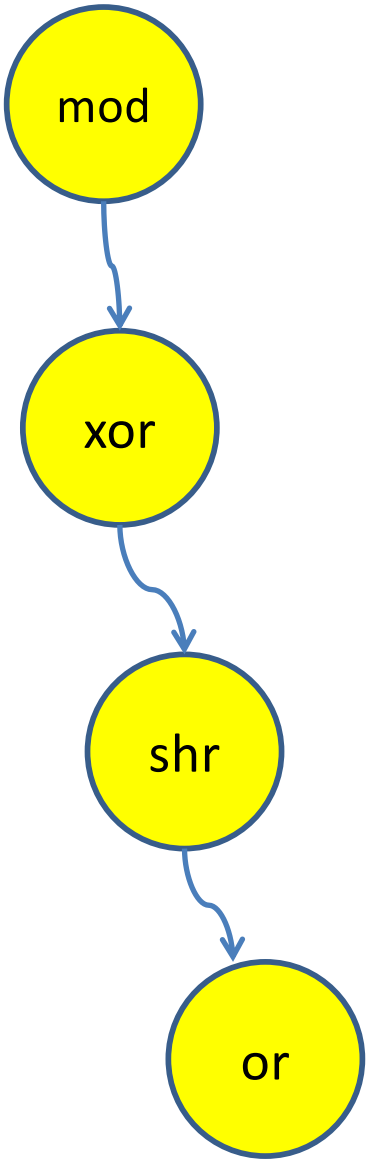
Dependency Constraints



- In this example, the subtract can only happen after the add and shift
- $X_{sub} - X_{add} >= 0$
- $X_{sub} - X_{shift} >= 0$

- Hence the name *difference constraints*

Handling Clock Period Constraints



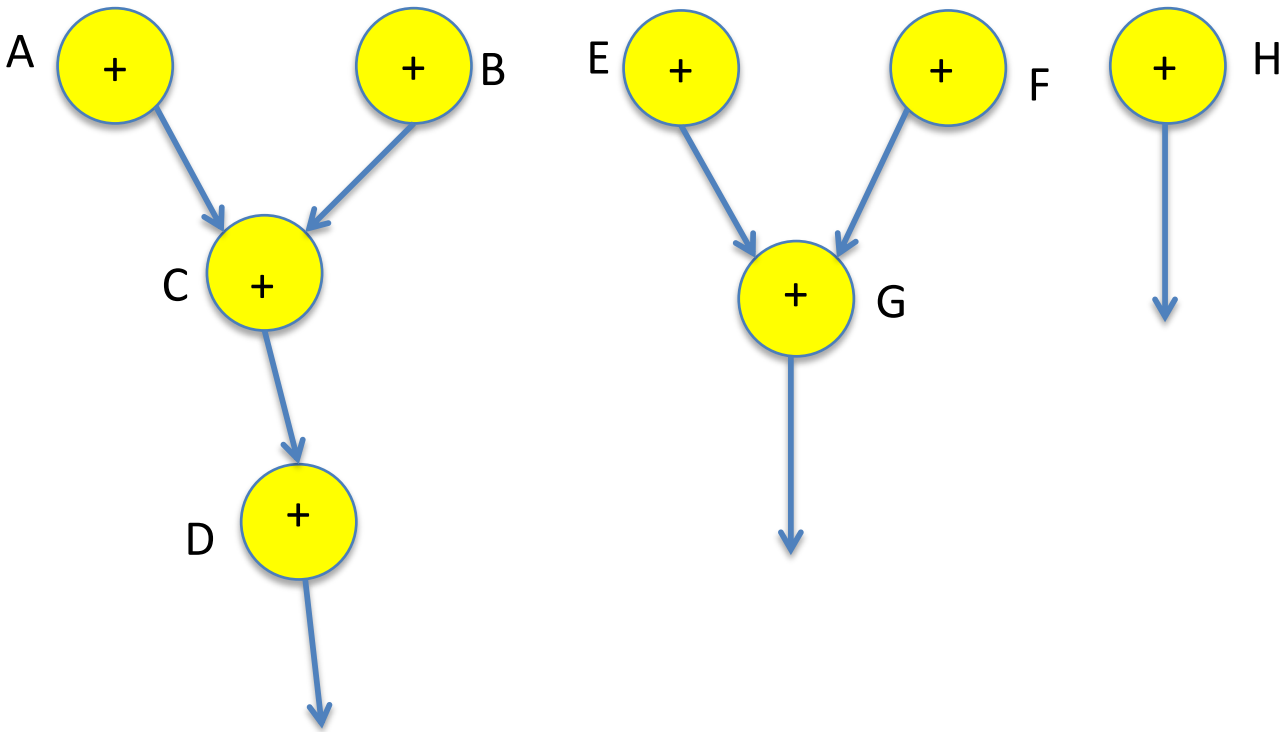
- Target period: P (e.g., 10 ns)
- For each chain of dependant operations in DFG, find the path delay D
  - E.g.: D from mod -> or = 23 ns.
- Compute: R = *ceiling*(D/P) - 1
  - E.g.: R = 2
- Add the *difference constraint*:
  - $X_{or} - X_{mod} >= 2$

Resource Constraints

- Restriction on # of operations of a given type that can execute in a cycle
- Why we need it?
  - Want to use dual-port RAMs in FPGA
    - Allow up to 2 load/store operations in a cycle
  - Floating point
    - Do not want to instantiate many FP cores of a given type, probably just one
    - Scheduling must honour # of FP cores available

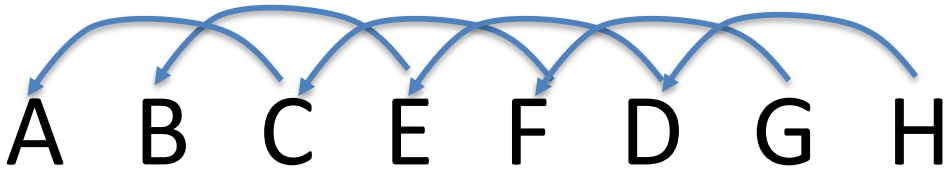
Resource Constraints in SDC

- Res-constrained scheduling is NP-hard
- Implemented approach in [Cong & Zhang DAC2006]



Add SDC Constraints

- Generate a topological ordering of the resource-constrained operations



- We have K (=2) adders in the HW,
  start at position i (initially K) and create constraint between op i and op i-K:

  $X_C - X_A >= 1$
- Bump up i and repeat:

  $X_E - X_B >= 1$

Resulting schedule will have at most K adders in a cycle

ASAP Objective Function

- Minimize the sum of the variables:

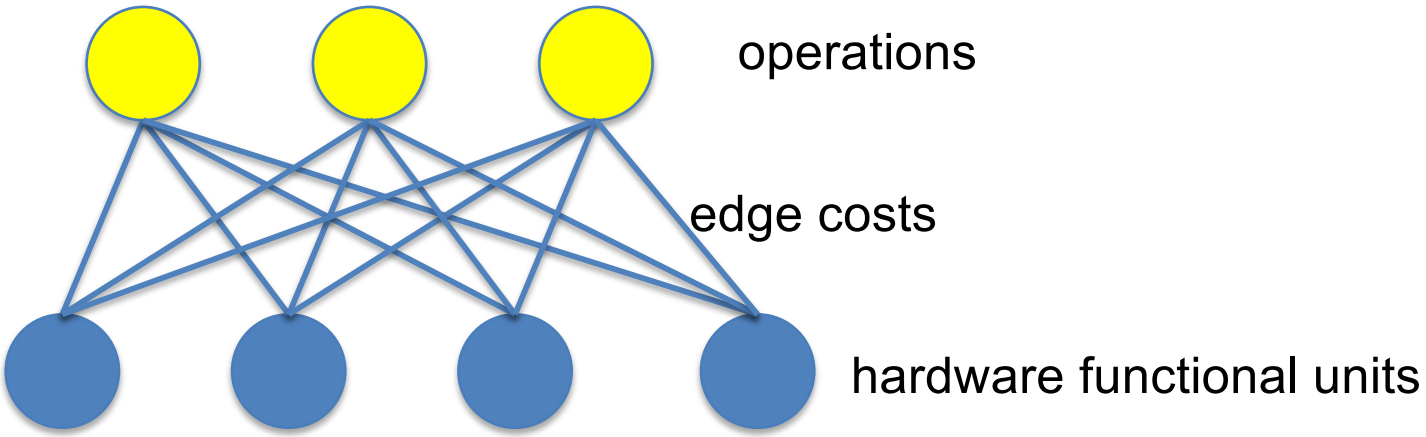
$$\text{minimize}(\text{f} = \sum_{\text{i} \in Ops} x_i)$$

- Operations will be scheduled as early as possible, subject to the constraints
- LP program solvable in polynomial time

High-Level Synthesis: Binding

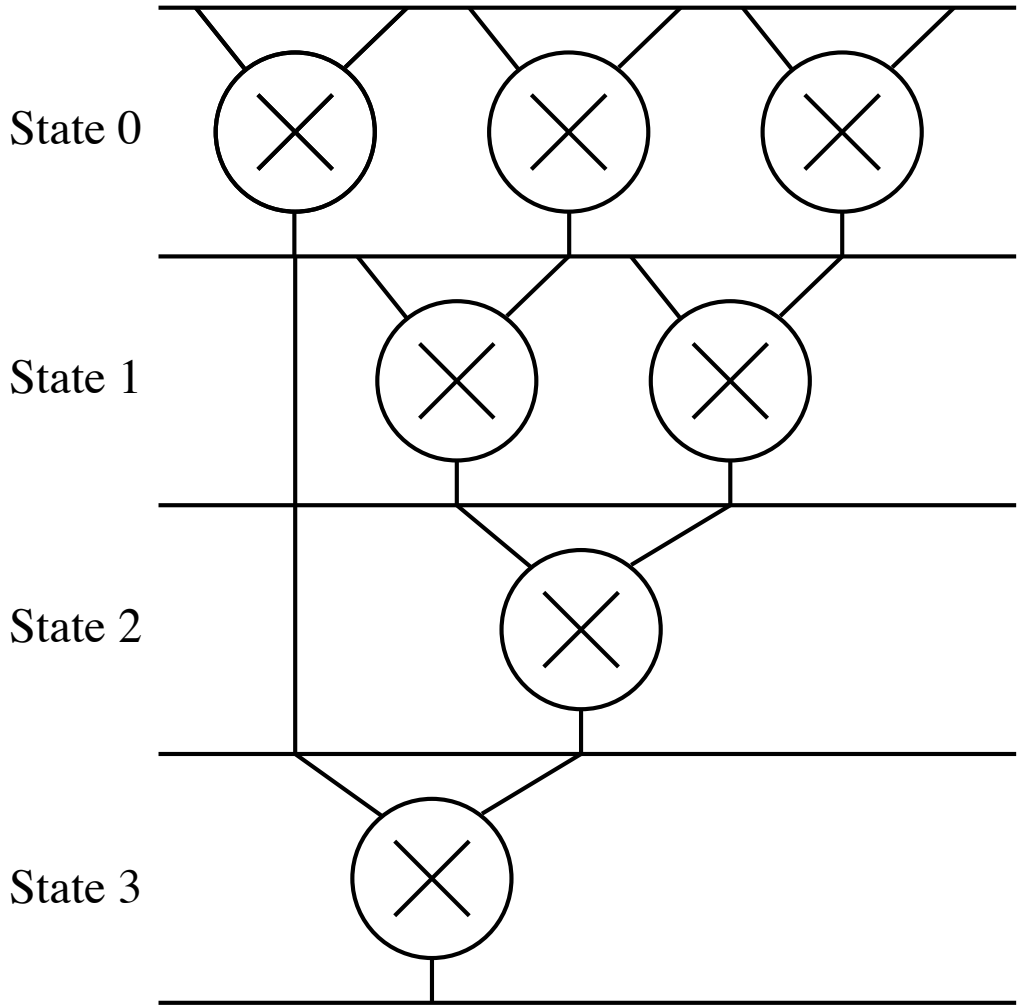
High-Level Synthesis: Binding

- Weighted bipartite matching-based binding
  - Huang, Chen, Lin, Hsu, "Data path allocation based on bipartite weighted matching". DAC 1990: 499-504.
- Finds the minimum weighted matching of a bipartite graph at each step
  - Solve using the Hungarian Method (polynomial)



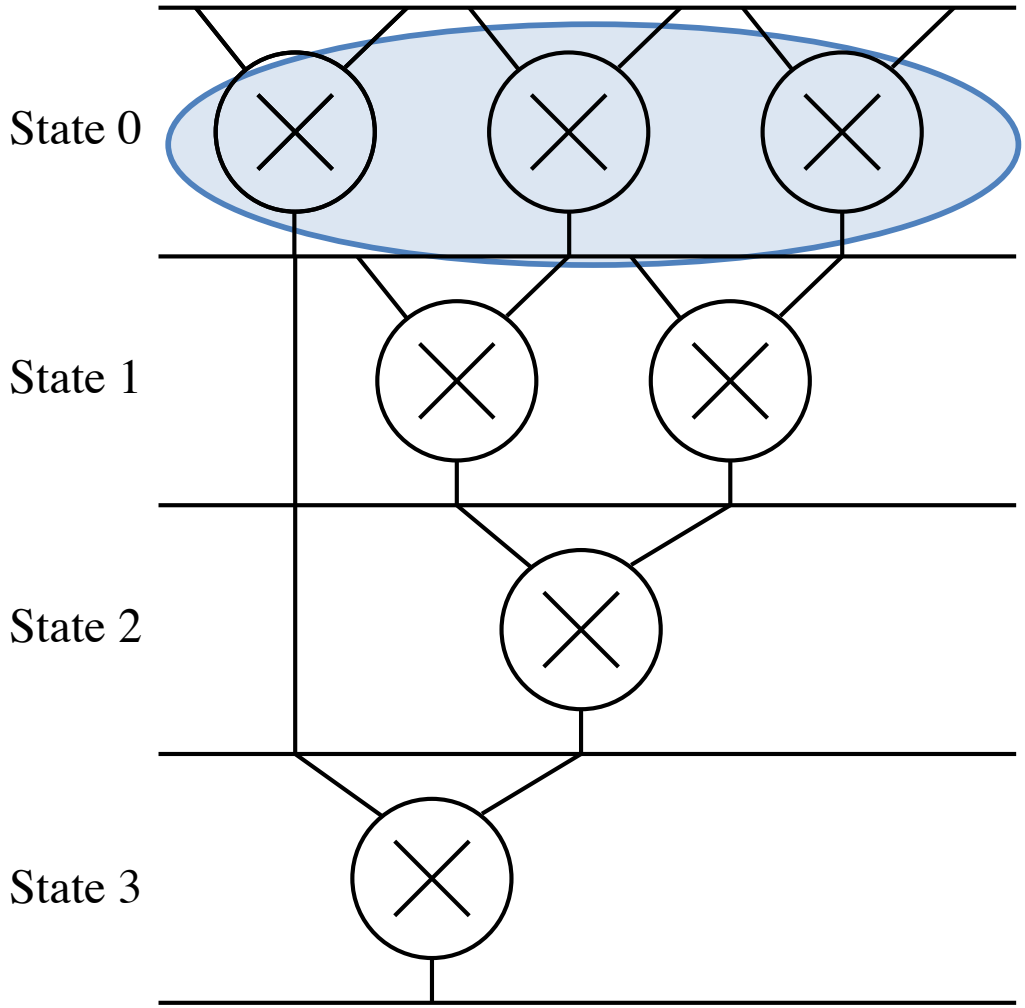
Binding

- Bind the following scheduled program



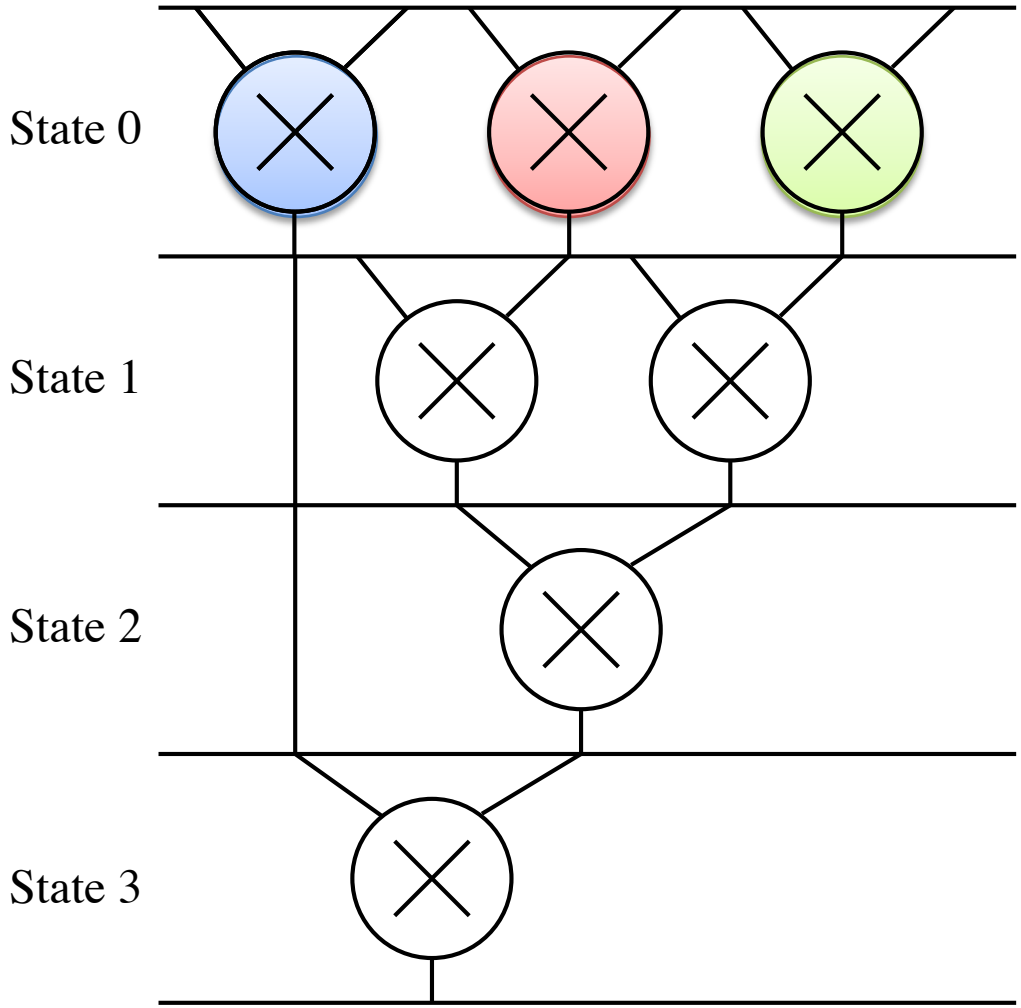
Binding

- Resource Sharing: requires 3 multipliers

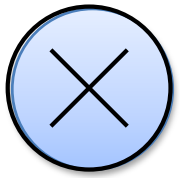


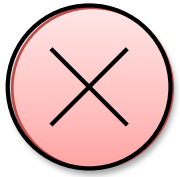
Binding

- Bind the first cycle



Functional Units

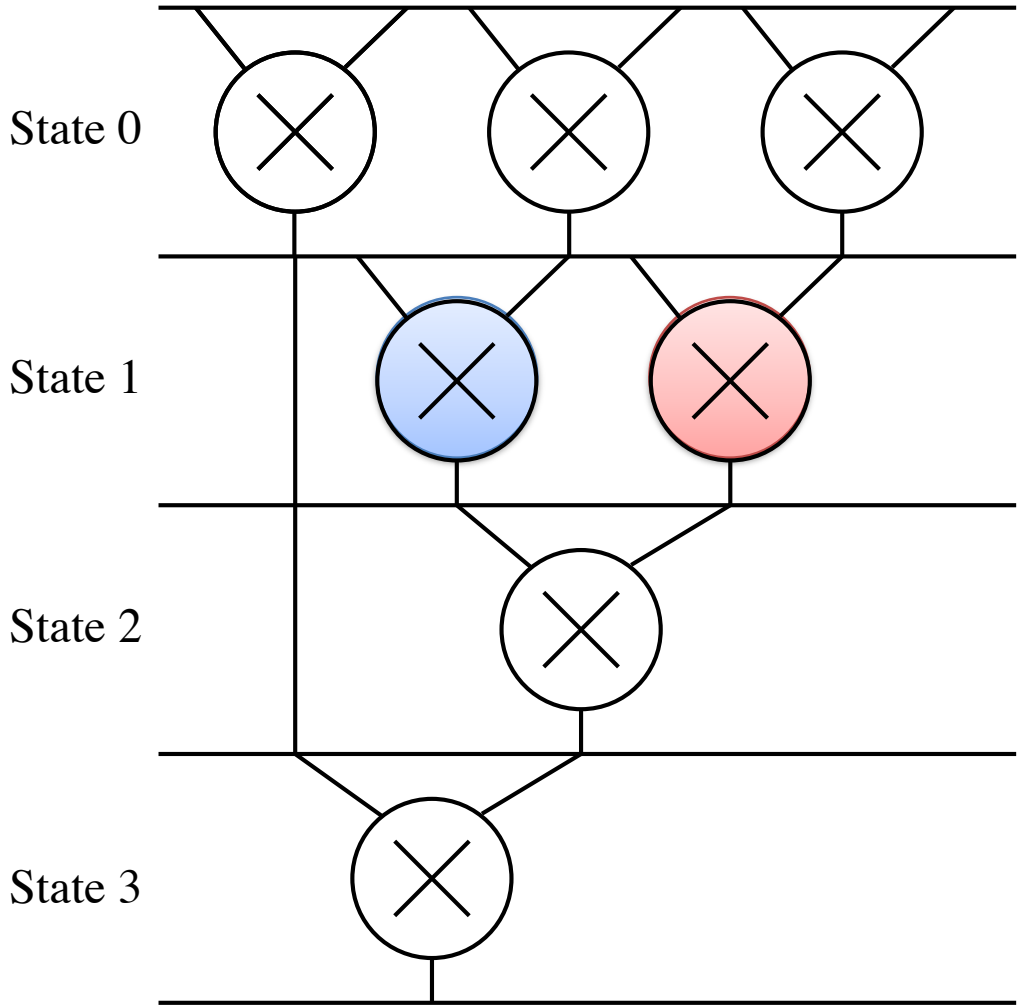

1


1

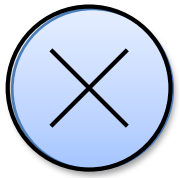

1

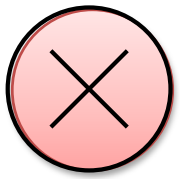
Binding

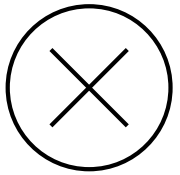
- Bind the second cycle



Functional Units

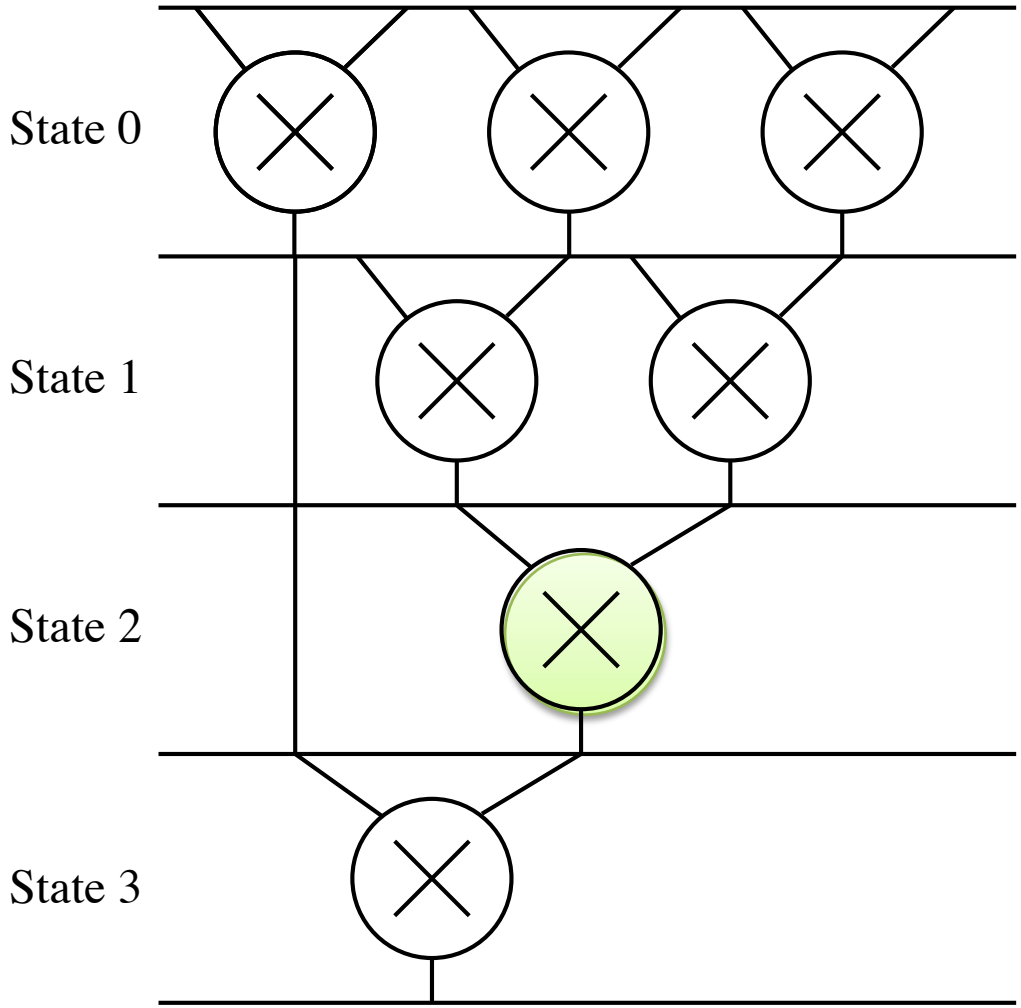

2


2

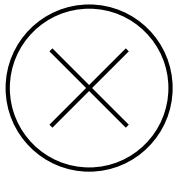

1

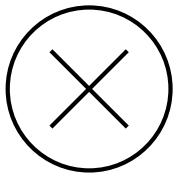
Binding

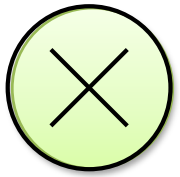
- Bind the third cycle



Functional Units

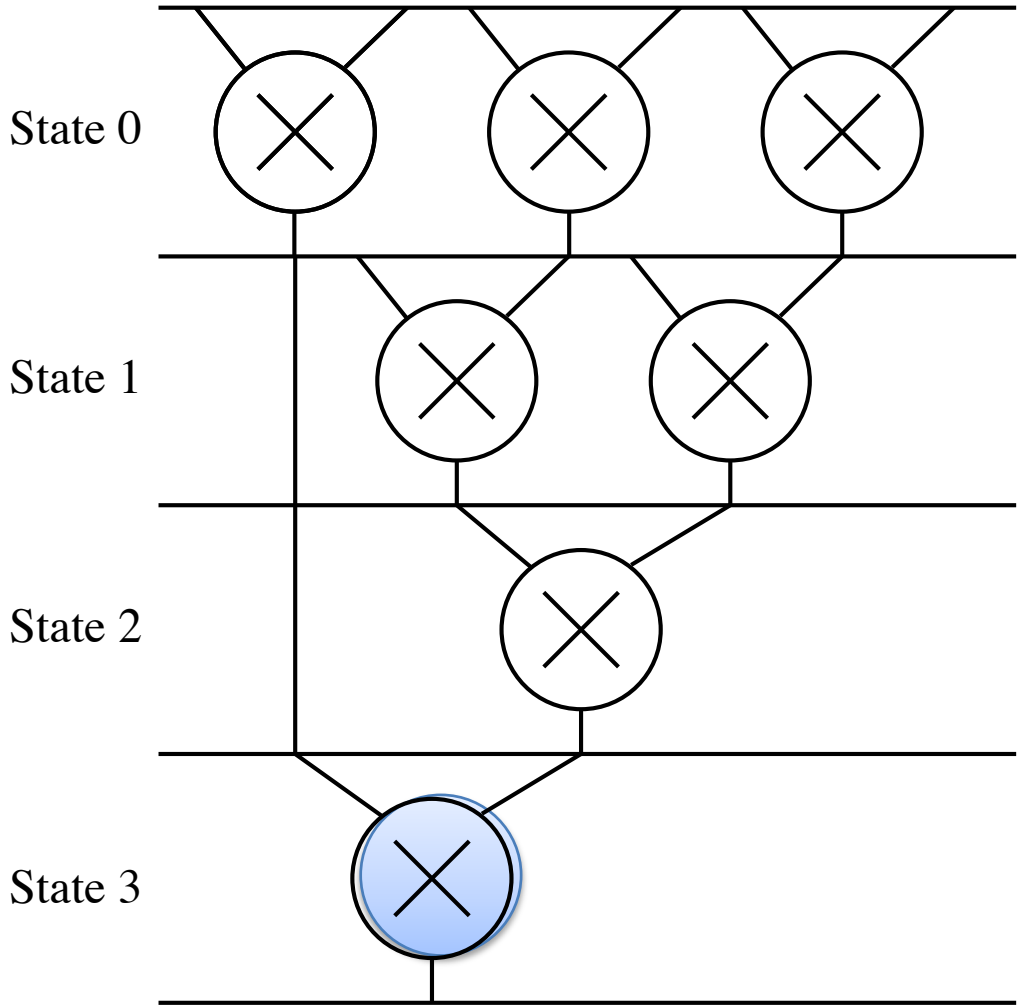

2


2

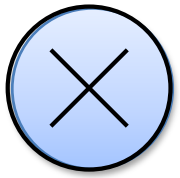

2

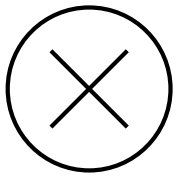
Binding

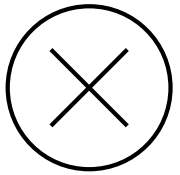
- Bind the fourth cycle



Functional Units

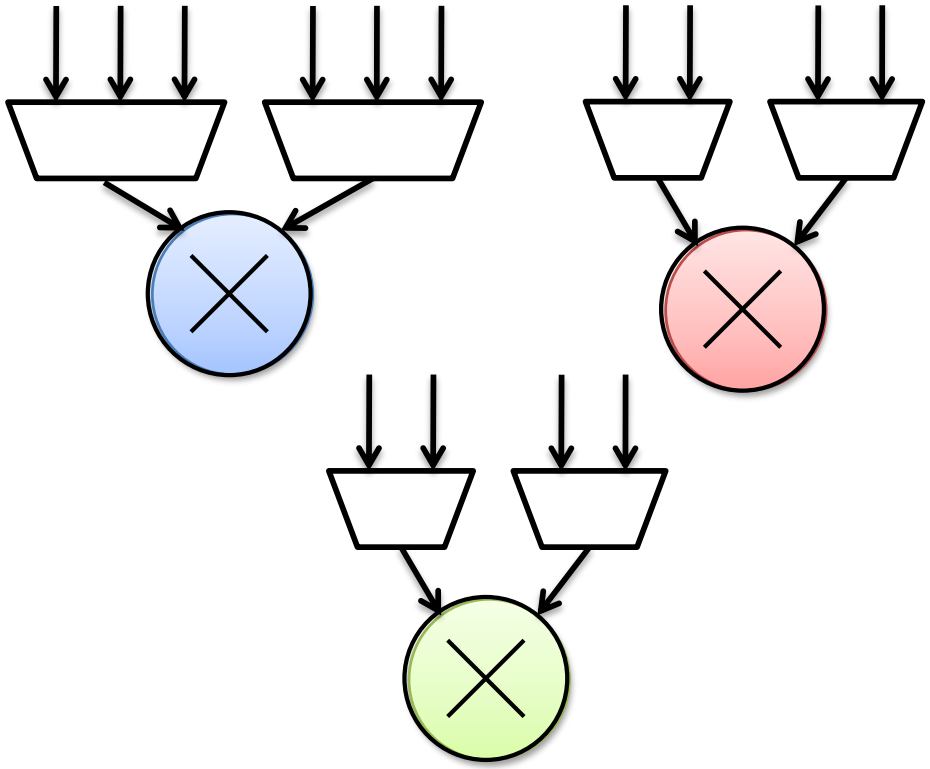

3


2

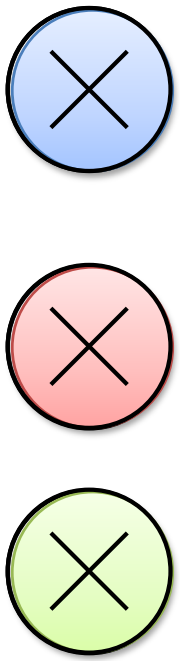

2

Binding

- Required Multiplexing:



Functional Units



Exploiting Parallelism

Generating Parallel Hardware

- Easy to extract instruction level parallelism using dependencies within a basic block
- **But** C code is inherently sequential and it is difficult to extract higher level parallelism
- LegUp provides two ways to generate parallel hardware
  - Loop pipelining
  - Pthreads/OpenMP

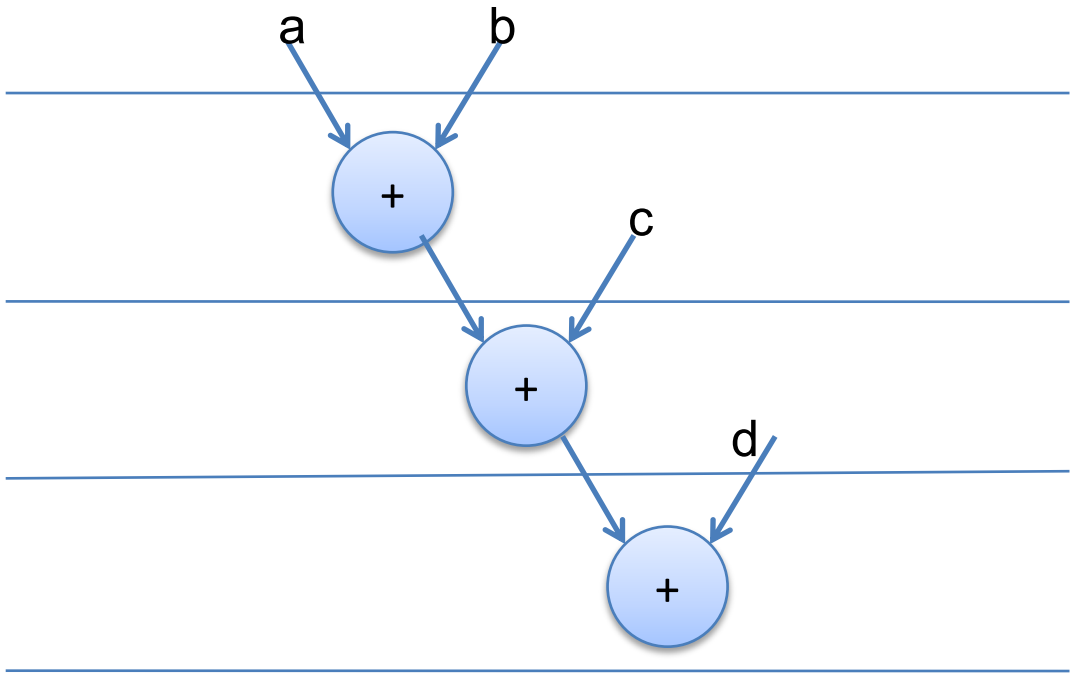
Loop Pipelining – Loop-Level Parallelism

Loop Pipelining Example

```
for (int i = 0; i < N; i++)
      sum[i] = a + b + c + d
```

Loop Pipelining Example

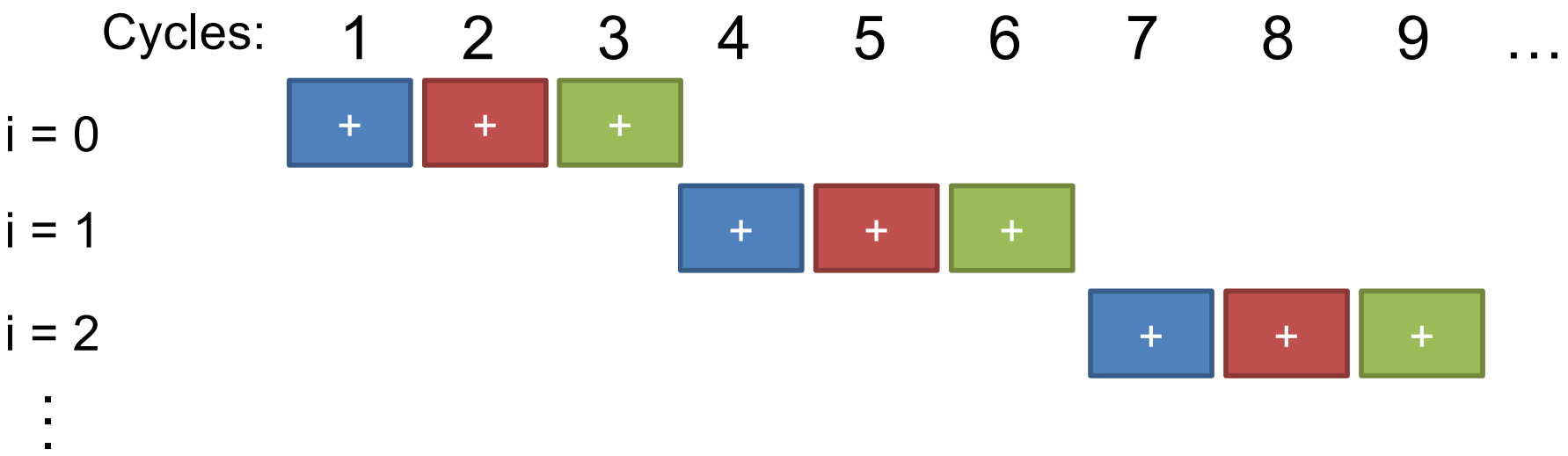
```
for (int i = 0; i < N; i++)
      sum[i] = a + b + c + d
```



Loop Pipelining Example

```
for (int i = 0; i < N; i++)
      sum[i] = a + b + c + d
```

1. Sequential Execution



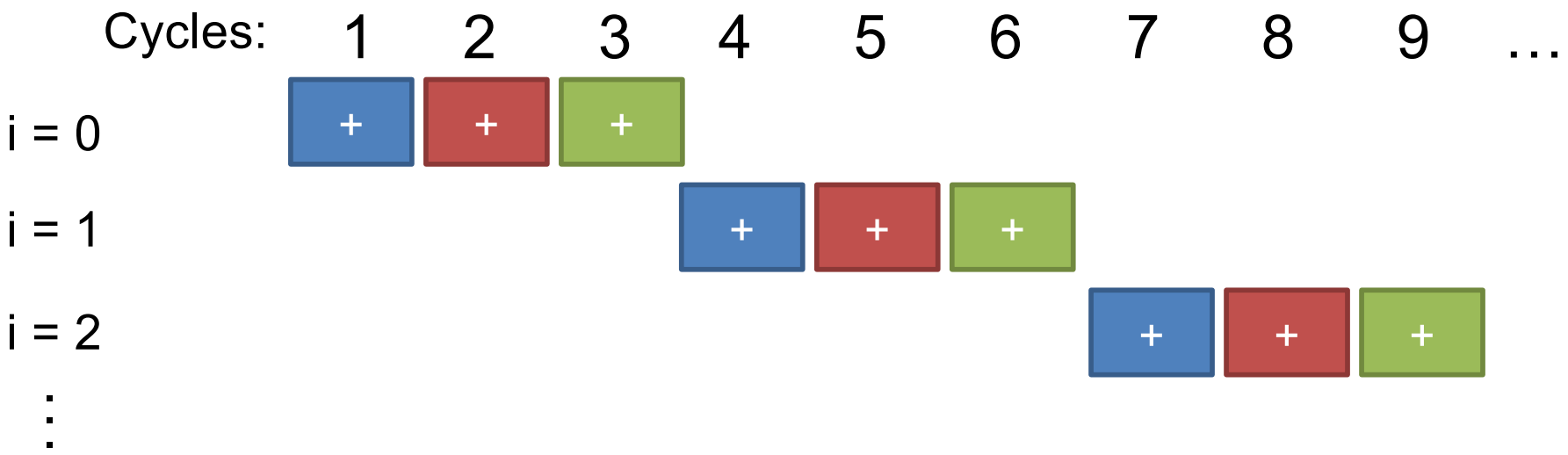
Loop Pipelining Example

```
for (int i = 0; i < N; i++)
    sum[i] = a + b + c + d
```

Not efficient!

- 3 Cycles/Iteration
- Total Cycles: 3N
- Adders: 3
- Adder Utilization: 33%

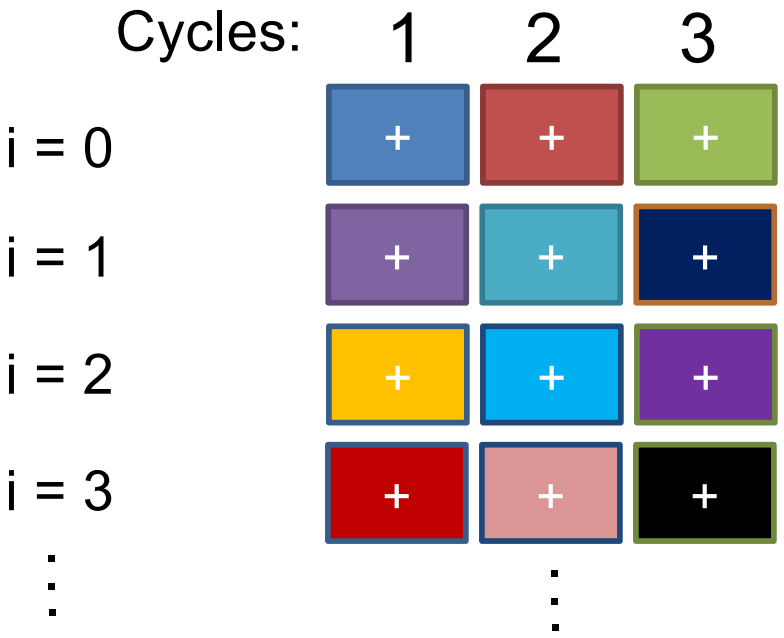
1. Sequential Execution



Loop Pipelining Example

```
for (int i = 0; i < N; i++)
      sum[i] = a + b + c + d
```

2. Parallel Execution : Loop unrolling



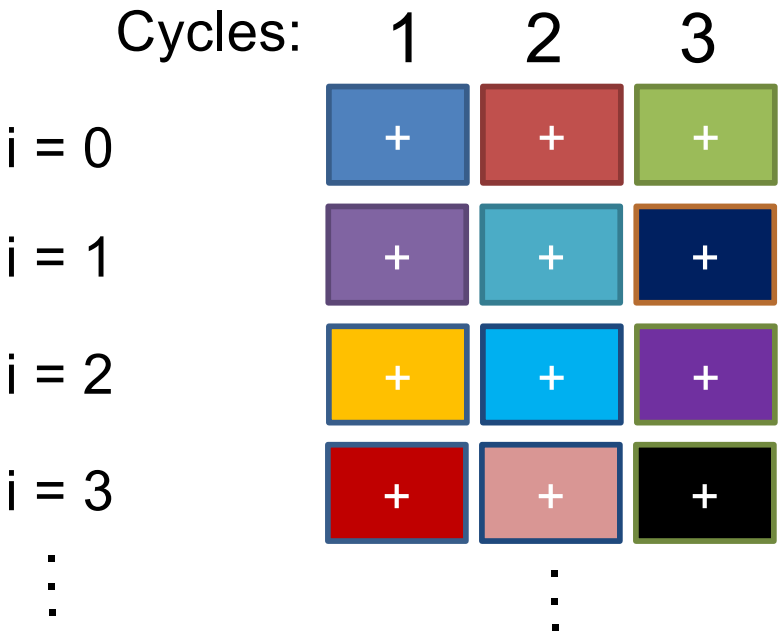
Loop Pipelining Example

```
for (int i = 0; i < N; i++)
    sum[i] = a + b + c + d
```

Not efficient!

- 3 Cycles/Iteration
- Total Cycles: 3
- Adders: 3N
- Adder Utilization: 33%

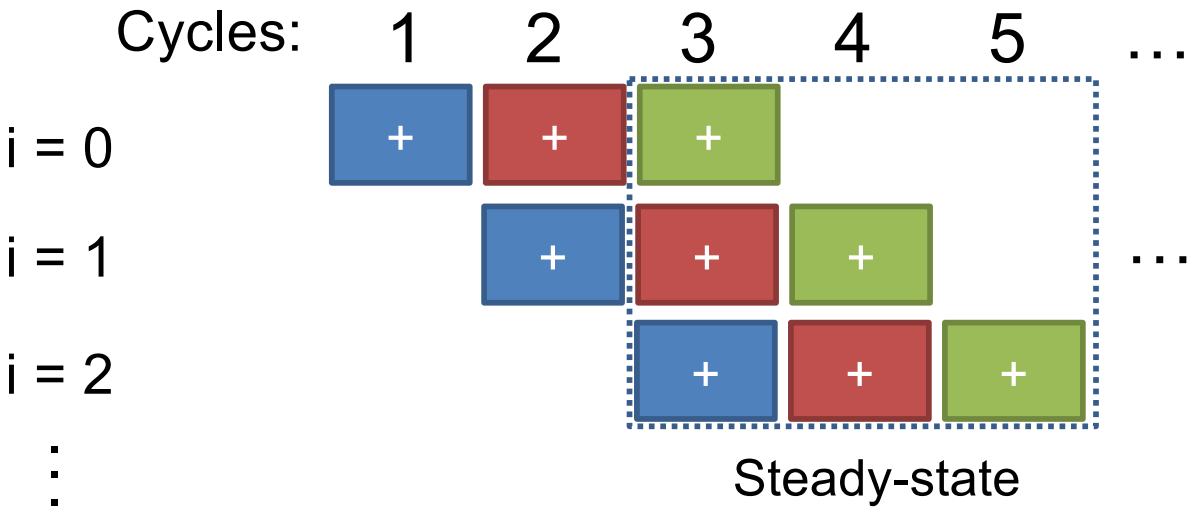
2. Parallel Execution : Loop unrolling



Loop Pipelining Example

```
for (int i = 0; i < N; i++)
      sum[i] = a + b + c + d
```

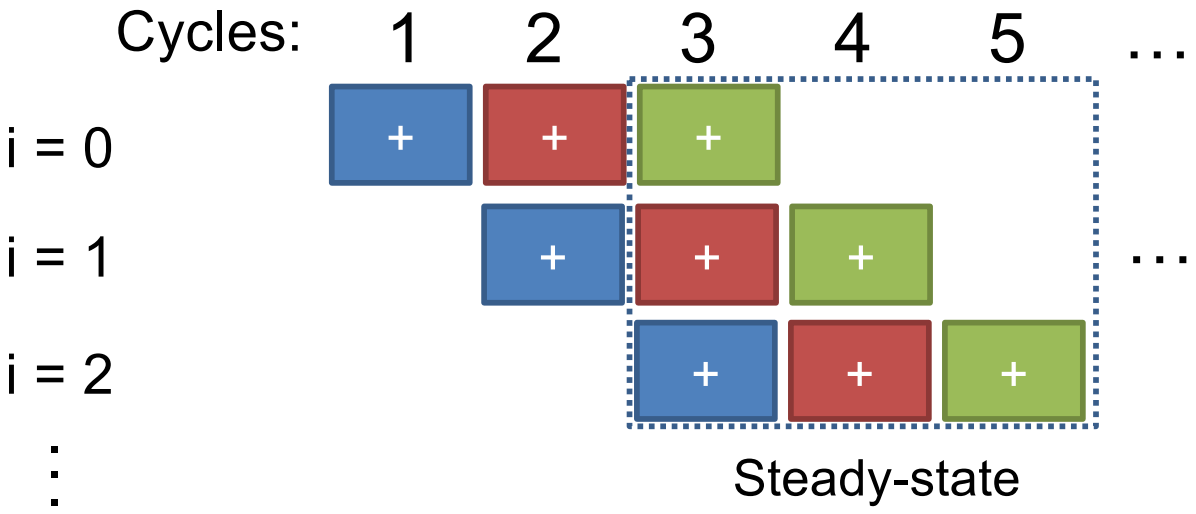
3. Parallel Execution : Loop pipelining



Loop Pipelining Example

```
for (int i = 0; i < N; i++)
    sum[i] = a + b + c + d
```

3. Parallel Execution : Loop pipelining



Efficient!

- 1 Cycles/Iteration (steady-state)
- Total Cycles: N+2
- Adders: 3
- Adder Utilization: 100%

Loop Pipelining

- Overlap execution of adjacent loop iterations
- Can be combined with loop unrolling
- HLS tools typically use an algorithm called:
  - Iterative Modulo Scheduling

Loop Pipelining Example

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

- Each iteration requires:
  - 2 loads from memory
  - 1 store
- No dependencies between iterations

Loop Pipelining Example

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

- Cycle latency of operations:
  - Load: 2 cycles
  - Store: 1 cycle
  - Add: 1 cycle
- Single memory port

LLVM Instructions

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

```
%i.04 = phi i32 [ 0, %bb.nph ],
                [ %3, %bb ]
%scevgep5 = getelementptr
            %b, %i.04
%0 = load %scevgep5
%scevgep6 = getelementptr
            %c, %i.04
%1 = load %scevgep6
%2 = add nsw i32 %1, %0
%scevgep =   getelementptr
            %a, %i.04
store %2, %scevgep
%3 = add %i.04, 1
%exitcond = eq %3, 100
br %exitcond, %bb2, %bb
```

LLVM Instructions

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

```
%i.04 = phi i32 [ 0, %bb.nph ],
                [ %3, %bb ]
%scevgep5 = getelementptr
                %b, %i.04
%0 = load %scevgep5
%scevgep6 = getelementptr
                %c, %i.04
%1 = load %scevgep6
%2 = add nsw i32 %1, %0
%scevgep =   getelementptr
                %a, %i.04
store %2, %scevgep
%3 = add %i.04, 1
%exitcond = eq %3, 100
br %exitcond, %bb2, %bb
```

LLVM Instructions

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

```
%i.04 = phi i32 [ 0, %bb.nph ],
                [ %3, %bb ]
%scevgep5 = getelementptr
                %b, %i.04
%0 = load %scevgep5
%scevgep6 = getelementptr
                %c, %i.04
%1 = load %scevgep6
%2 = add nsw i32 %1, %0
%scevgep =   getelementptr
                %a, %i.04
store %2, %scevgep
%3 = add %i.04, 1
%exitcond = eq %3, 100
br %exitcond, %bb2, %bb
```

LLVM Instructions

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

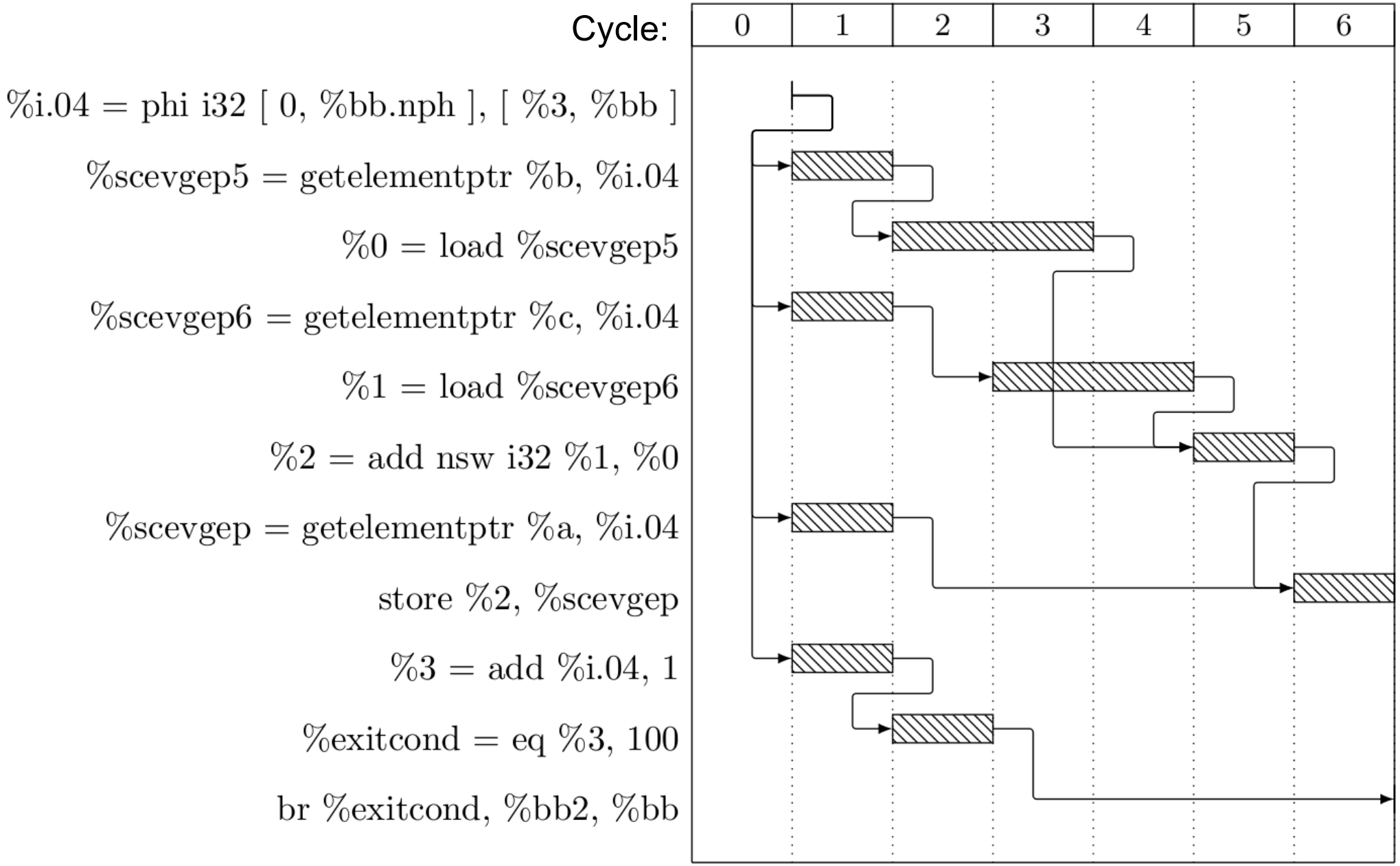
```
%i.04 = phi i32 [ 0, %bb.nph ],
                [ %3, %bb ]
%scevgep5 = getelementptr
                %b, %i.04
%0 = load %scevgep5
%scevgep6 = getelementptr
                %c, %i.04
%1 = load %scevgep6
%2 = add nsw i32 %1, %0
%scevgep =   getelementptr
                %a, %i.04
store %2, %scevgep
%3 = add %i.04, 1
%exitcond = eq %3, 100
br %exitcond, %bb2, %bb
```

LLVM Instructions

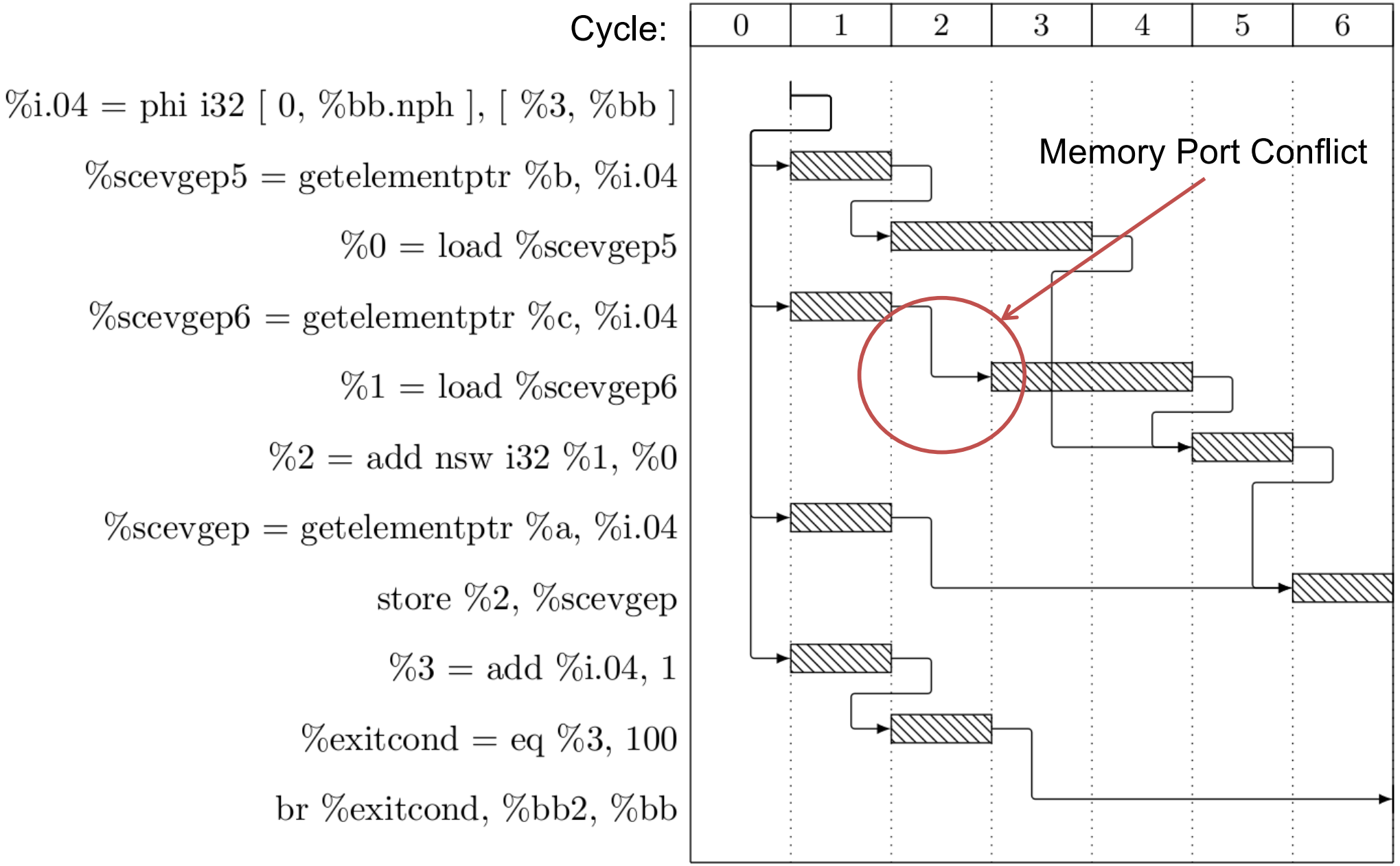
```
for (int i = 0; i < N; i++) {
    a[i] = b[i] + c[i]
}
```

```
%i.04 = phi i32 [ 0, %bb.nph ],
                [ %3, %bb ]
%scevgep5 = getelementptr
            %b, %i.04
%0 = load %scevgep5
%scevgep6 = getelementptr
            %c, %i.04
%1 = load %scevgep6
%2 = add nsw i32 %1, %0
%scevgep =   getelementptr
            %a, %i.04
store %2, %scevgep
%3 = add %i.04, 1
%exitcond = eq %3, 100
br %exitcond, %bb2, %bb
```

Scheduling LLVM Instructions



Scheduling LLVM Instructions



Loop Pipelining Example

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

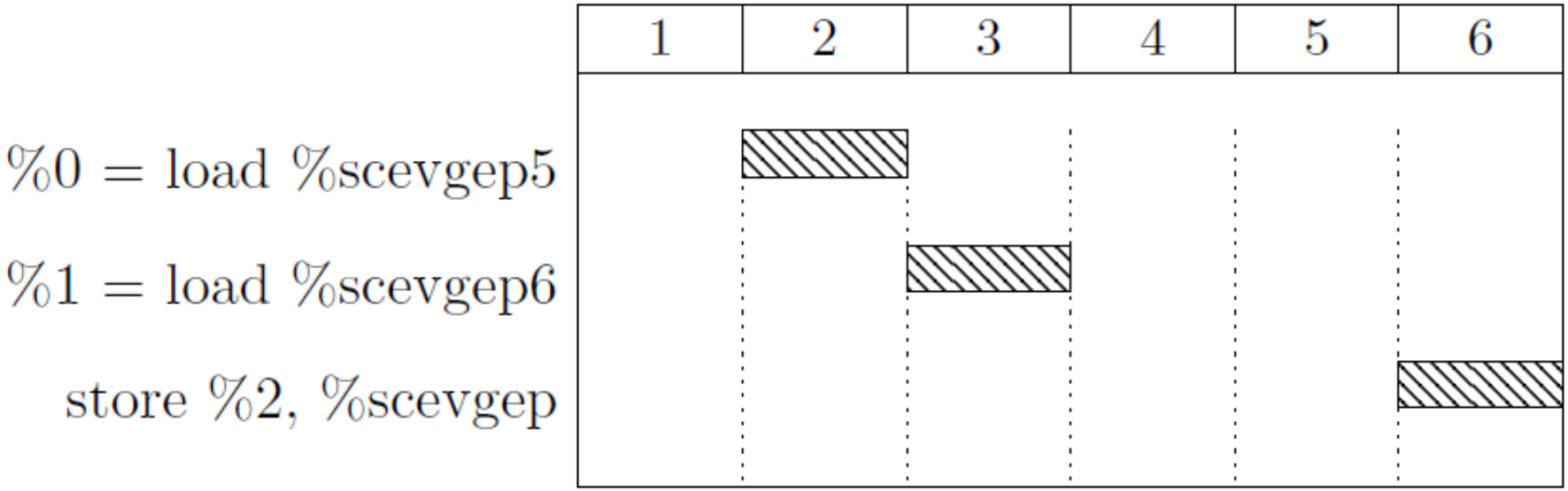
- **Initiation Interval (II)**
  - Constant time interval between starting successive iterations of the loop
- The loop requires 6 cycles per iteration (II=6)
- Can we do better?

Minimum Initiation Interval

- Resource minimum II:
  - Due to functional units
  - ResMII = $\frac{\text{Uses of functional unit}}{\text{\# of functional units}}$
- Recurrence minimum II:
  - Due to loop carried dependencies
- Minimum II = max(ResMII, RecMII)

Resource Constraints

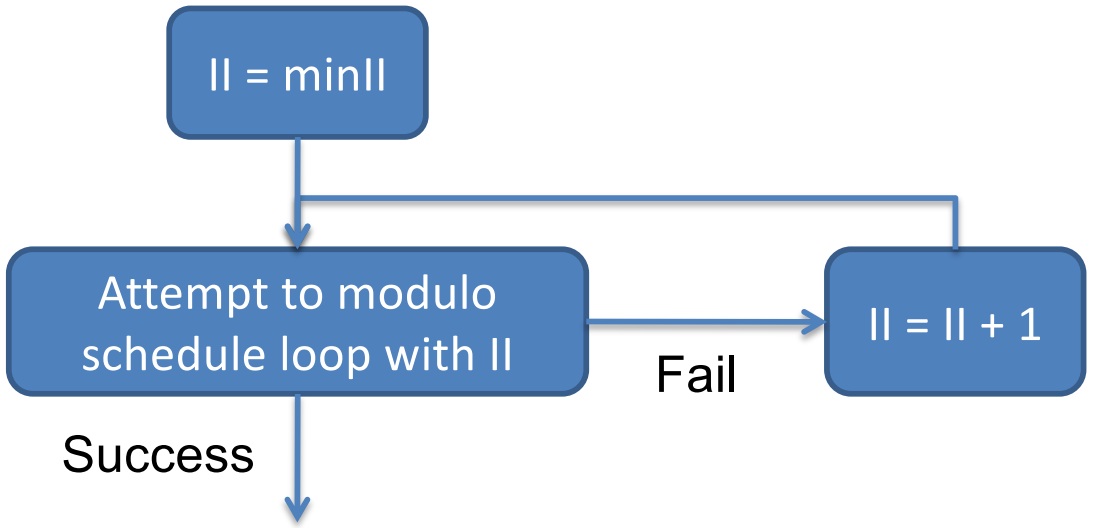
- Assume unlimited functional units (adders, …)
- Only constraint: **single** ported memory controller
- Reservation table:



- The resource minimum initiation interval is 3

Iterative Modulo Scheduling

- There are no loop carried dependencies so Minimum II = ResMII = 3
- Iterative: Not always possible to schedule the loop for minimum II



Iterative Modulo Scheduling

- Operations in the loop that execute in cycle:

  i

- Must also execute in cycles:

  i + k*II k = 0 to N-1

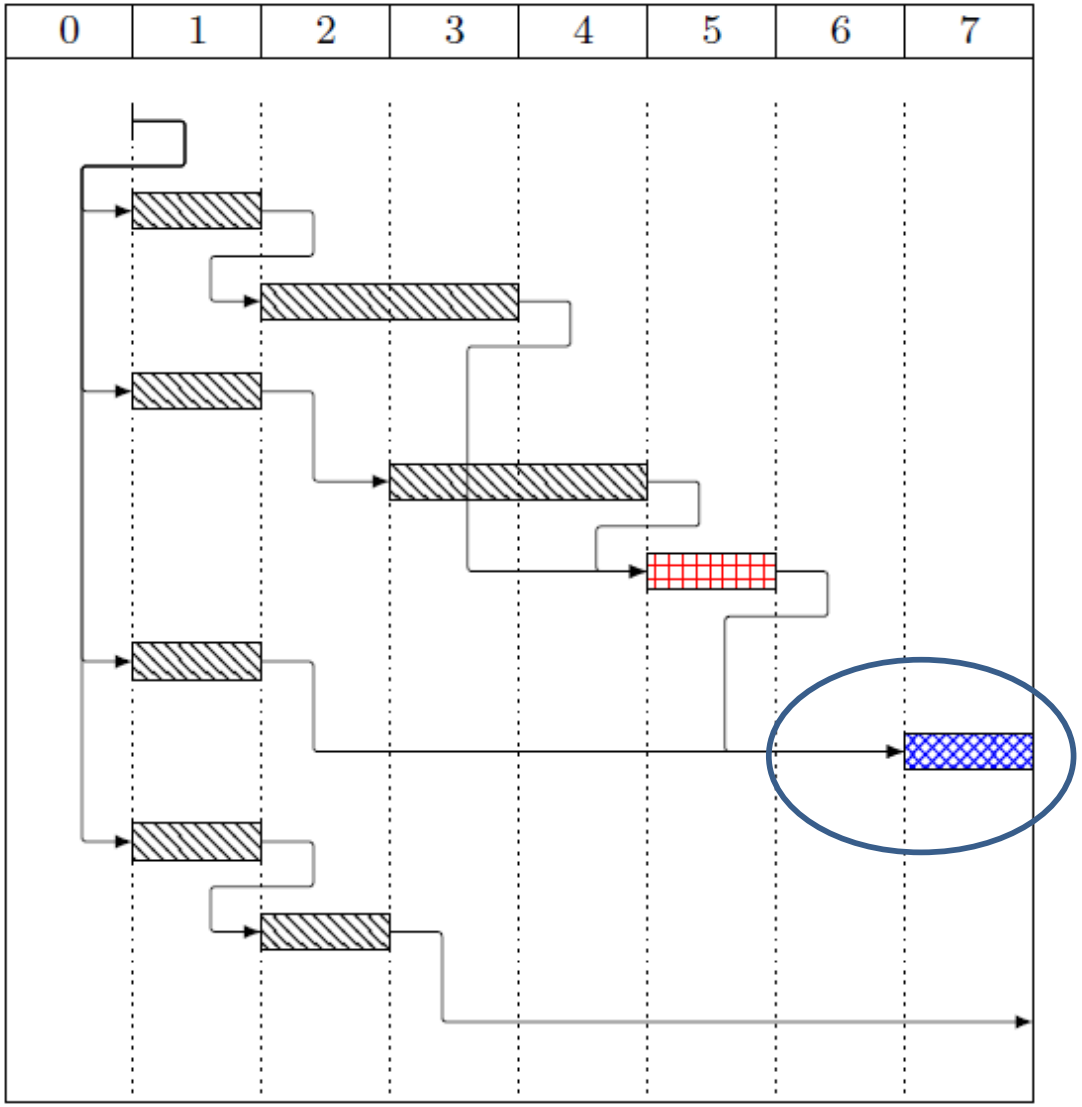
- Therefore to detect resource conflicts look in the reservation table under cycle:

  (i-1) **mod** II + 1

- Hence the name “modulo scheduling”

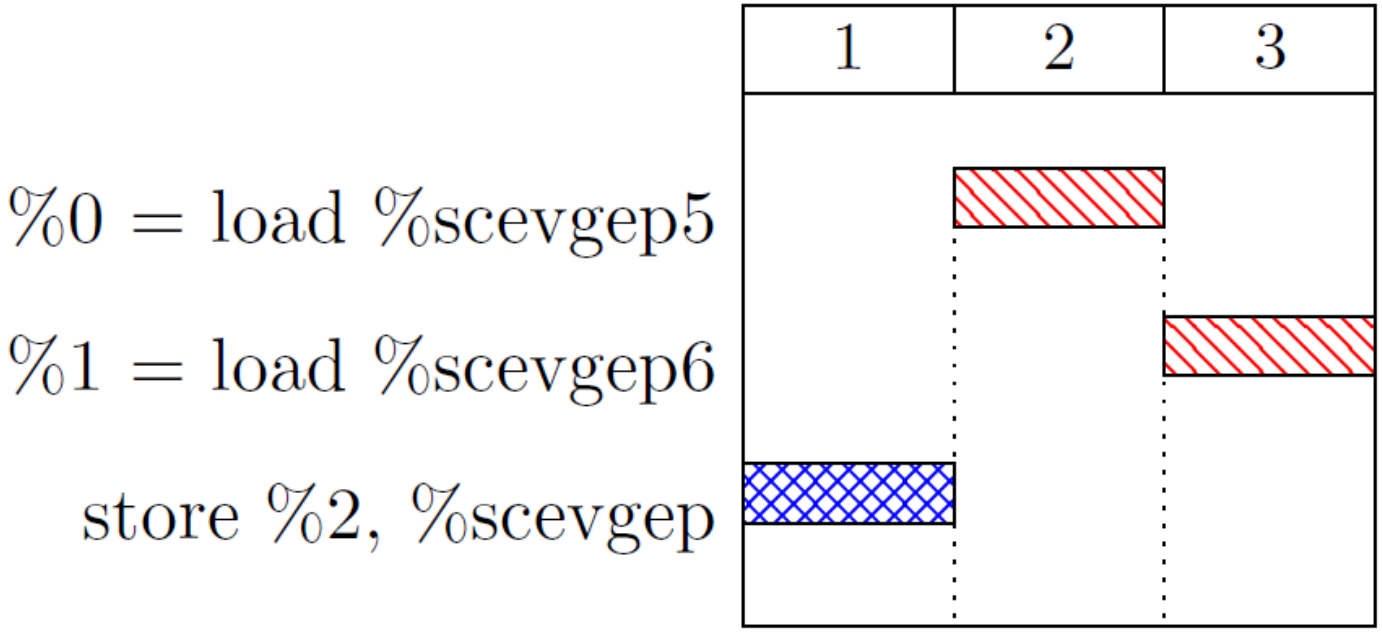
New Pipelined Schedule

```
%i.04 = phi i32 [ 0, %bb.nph ], [ %3, %bb ]
%scevgep5 = getelementptr %b, %i.04
%0 = load %scevgep5
%scevgep6 = getelementptr %c, %i.04
%1 = load %scevgep6
%2 = add nsw i32 %1, %0
%scevgep = getelementptr %a, %i.04
store %2, %scevgep
%3 = add %i.04, 1
%exitcond = eq %3, 100
br %exitcond, %bb2, %bb
```



Modulo Reservation Table

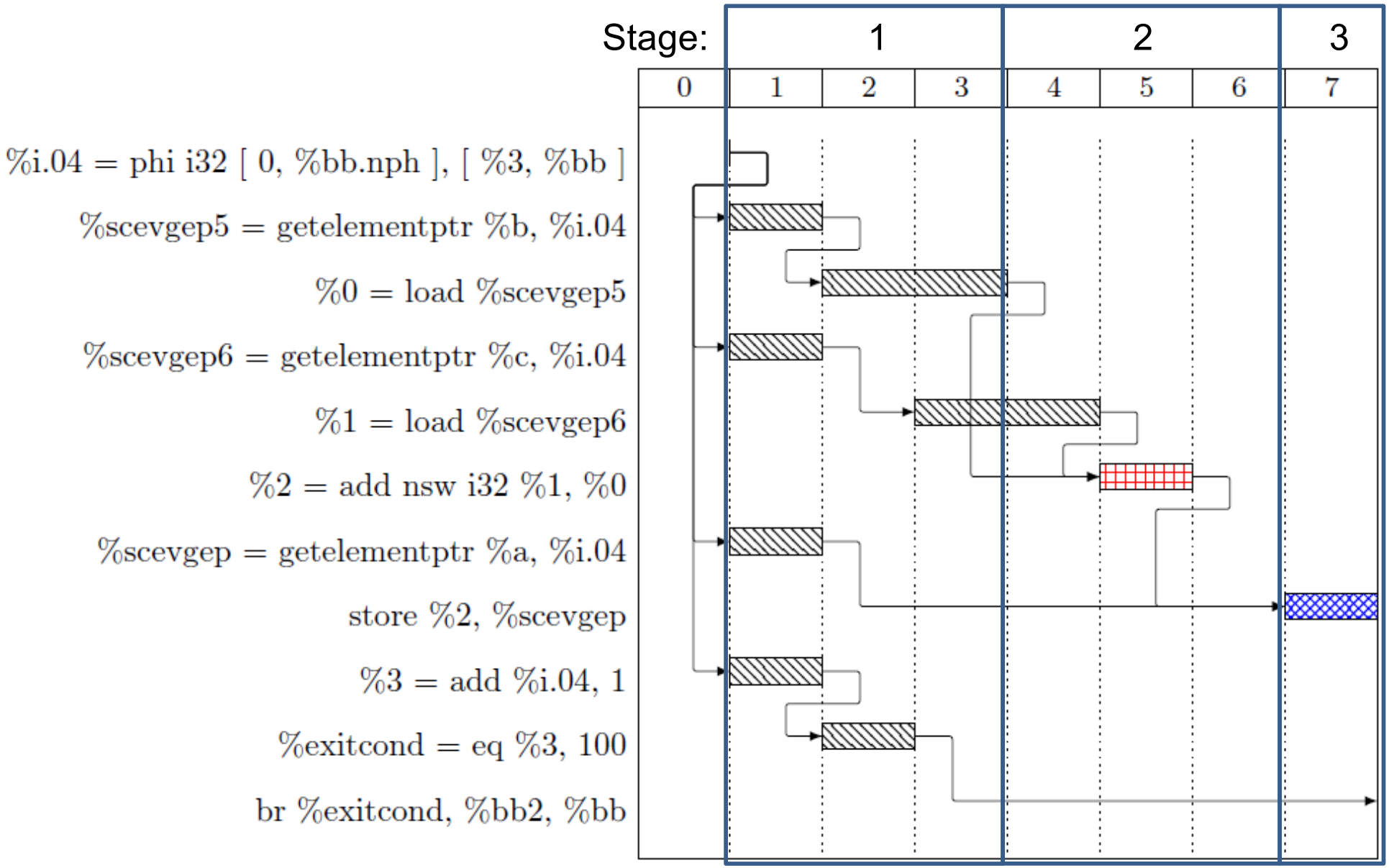
- Store couldn’t be scheduled in cycle 6
- Slot = (6-1) mod 3 + 1 = 3
- Already taken by an earlier load



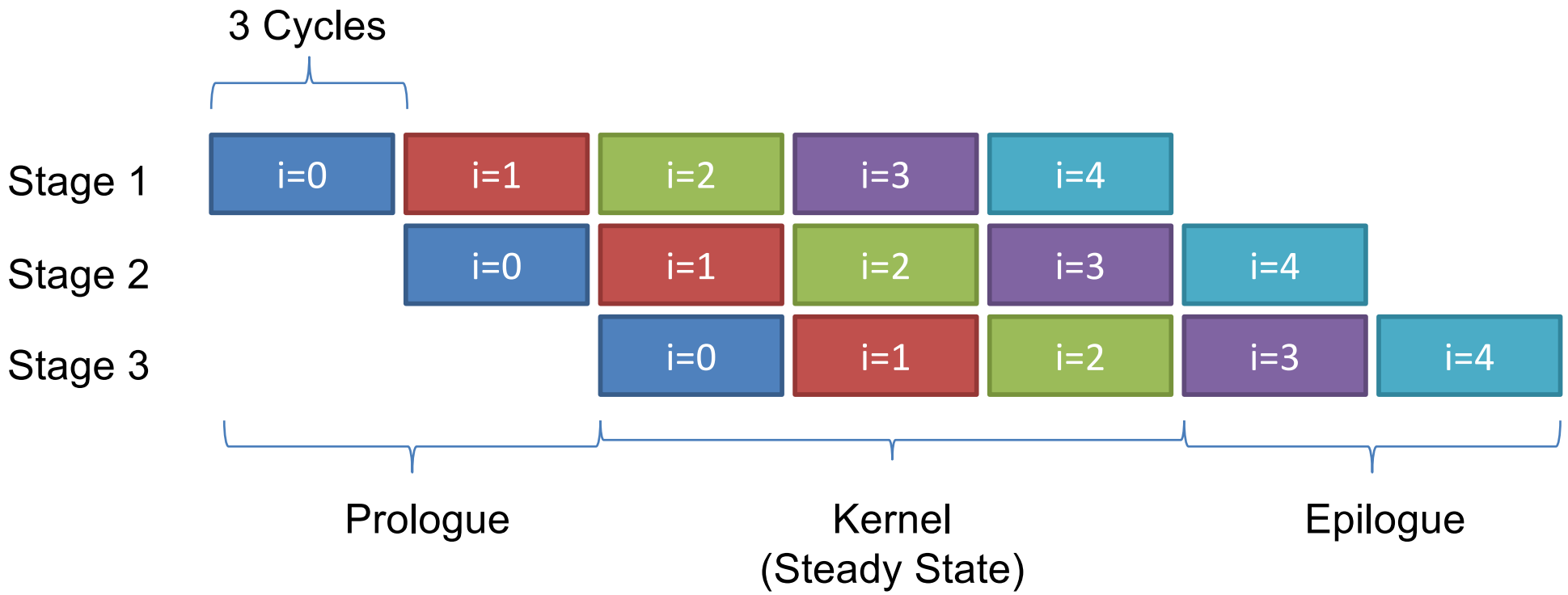
Iterative Modulo Scheduling

- Now we have a valid schedule for II=3
- We need to construct the loop **kernel**, **prologue**, and **epilogue**
- The loop kernel is what is executed when the pipeline is in steady state
  - The kernel is executed every II cycles
- First we divide the schedule into **stages** of II cycles each

Pipeline Stages

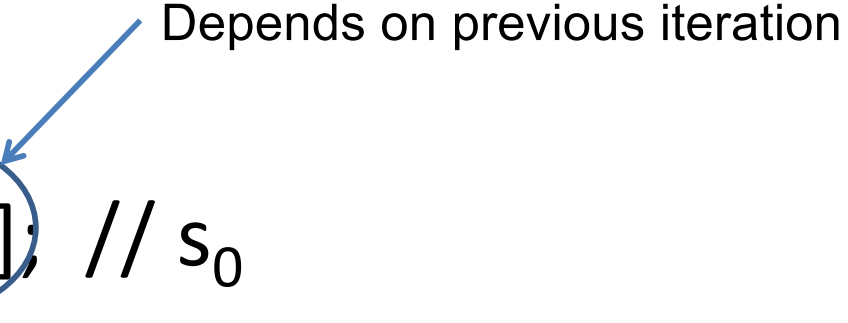


Pipelined Loop Iterations



Loop Dependencies

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        a[j] = b[i] + a[j-1];  // s0
```



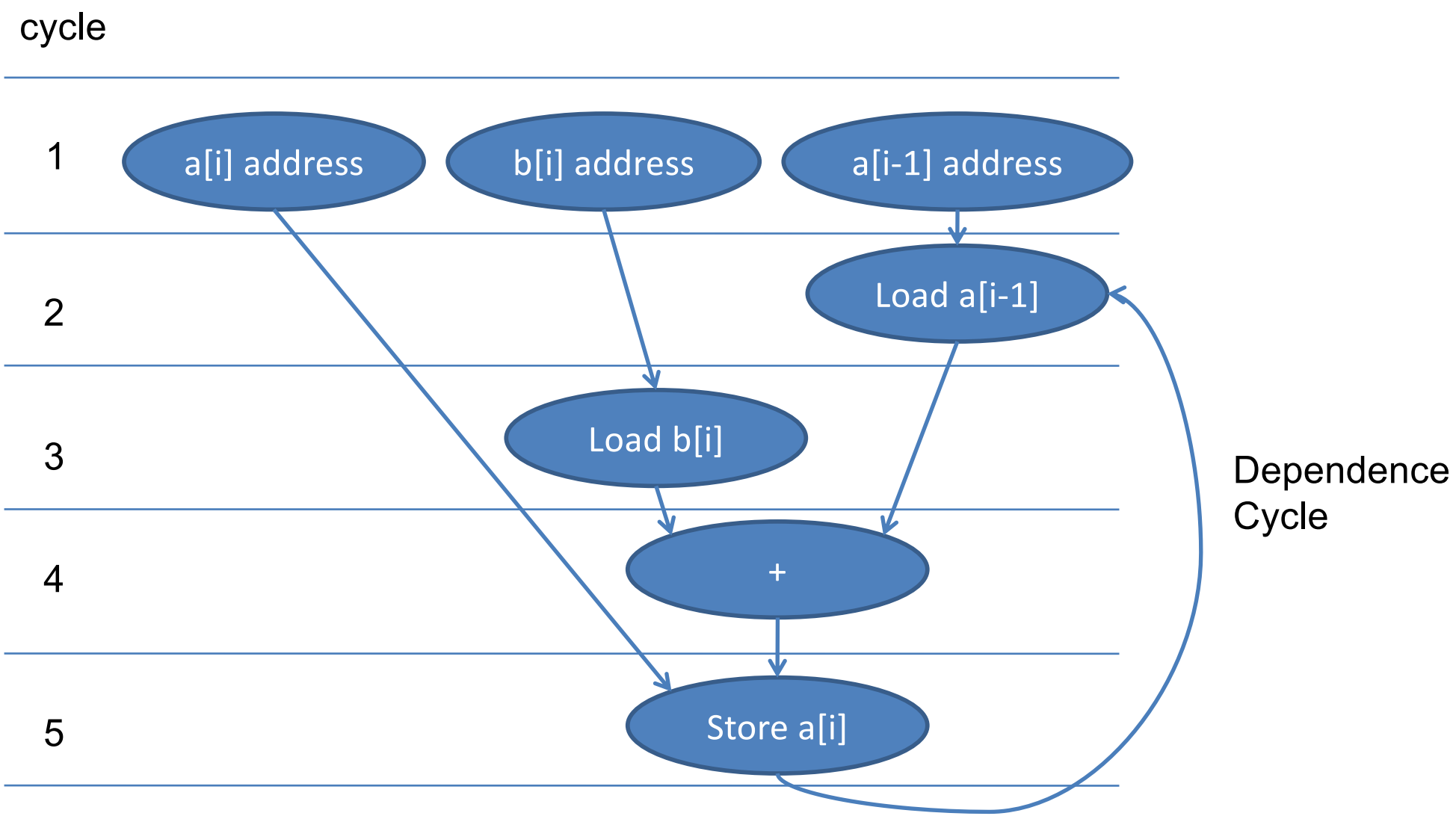
- Dependence distance: 1
- Induction variable: j

Recurrence Minimum II

```
for (i = 0; i < N; i++) {
   a[i] = b[i] + a[i-1];
}
```

- How does this loop dependence affect the minimum initiation interval?

Cycles in Dependence Graph



Recurrence Minimum II

```
for (j = 0; j < N; j++) {
   a[j] = b[i] + a[j-1];
}
```

- Dependence cycle has a length of 4
- Dependence distance is 1
- RecMII = $\frac{\text{Length of cycle in dependency graph}}{\text{Loop dependence distance}}$
- RecMII = 4/1 = 4

Loop Pipelining Summary

- Crucial concept in HLS to produce high-speed hardware
- Want II to be as **low** as possible. Why?
  - Total cycles spend in loop is approx.: N*II
- Ability to minimize II limited by:
  - Resource constraints
  - Cross-iterations dependencies
- Results impacted by way in which input program is structured

Spatial Parallelism

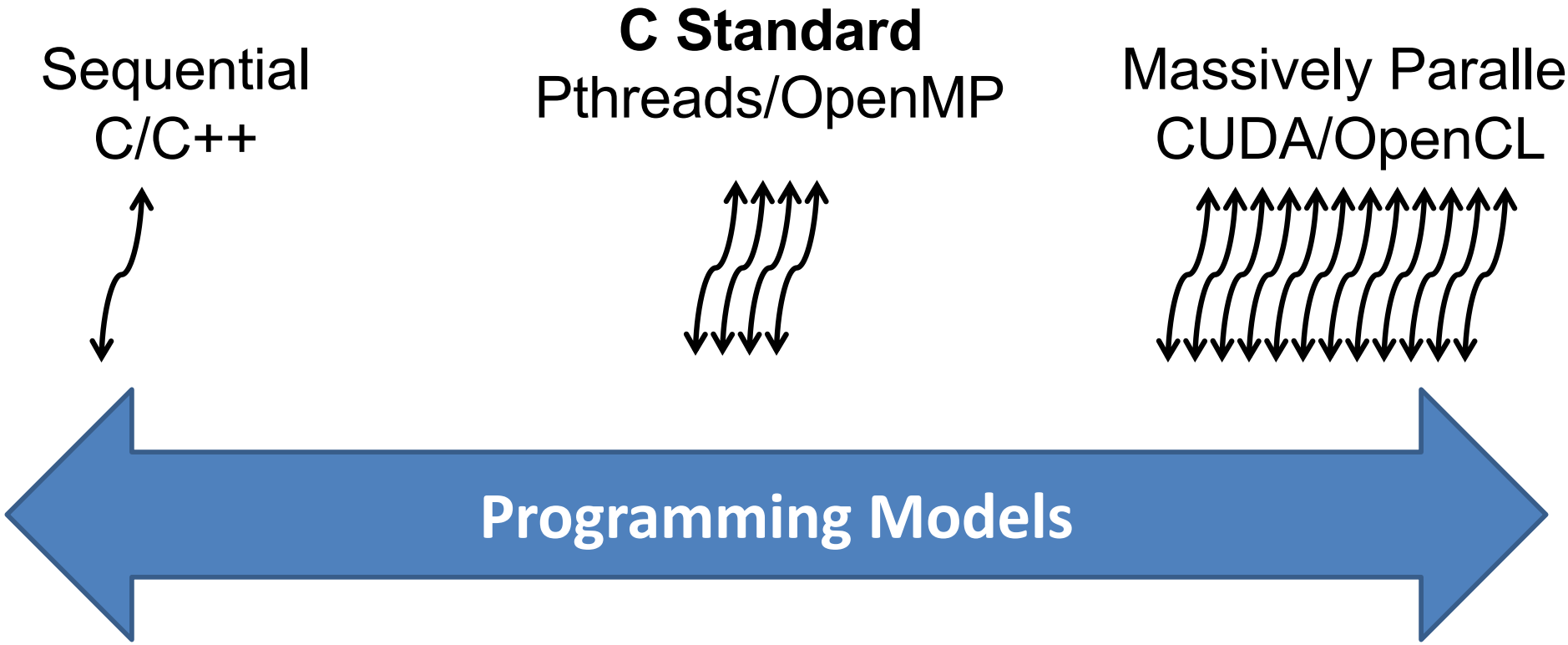
Motivation

- Speed benefits of HW arise from spatial parallelism
- Extracting parallelism from a sequential program is difficult
- Auto-parallelizing compilers do not work well!

- Easier to start from parallel code
- Pthreads/OpenMP can help!

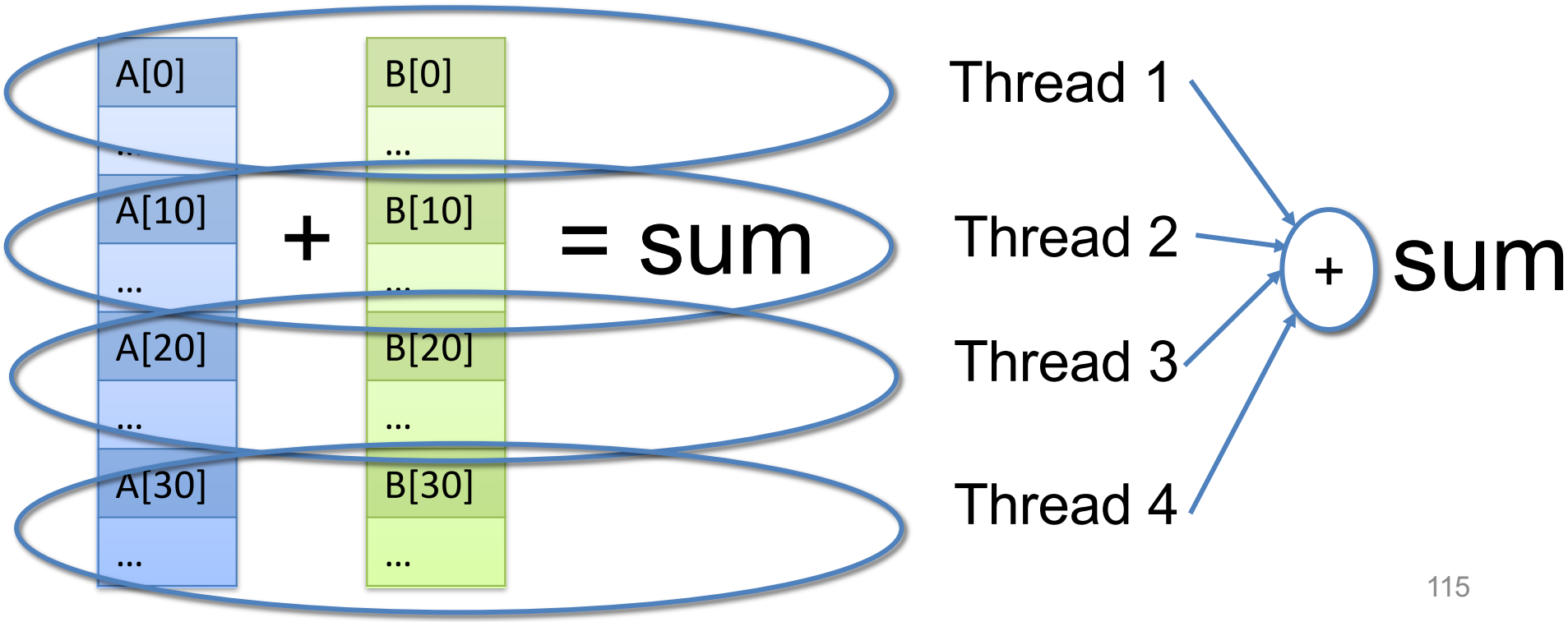
Programming Models

- LegUp has support for two standard parallel programming methodologies:
  - Pthreads and OpenMP



Pthreads/OpenMP

- Allows software engineer to express parallelism
- Multiple hardware accelerators running in parallel
- Each thread = one hardware accelerator
- Support for thread synchronization: mutexes and barriers



Pthreads Example

```
int main() {
   …
   for (i=0; i<4; i++) {
      pthread_create(&threads[i],
      NULL, add, &data[i]);
   }

   for (i=0; i<4; i++) {
      pthread_join(threads[i],
      (void**)&result[i]);
   }
   …
}
```

```
void *add(void *threadarg) {
   …
   for (i=startIdx; i<endidx; i++) {
      sum = A[i] + B[i];
   }
   …
}
```

Pthreads Example

```
int main() {
   …
   for (i=0; i<4; i++) {
      pthread_create(&threads[i],
      NULL, add, &data[i]);
   }

   for (i=0; i<4; i++) {
      pthread_join(threads[i],
      (void**)&result[i]);
   }
   …
}
```

```
void *add(void *threadarg) {
   …
   for (i=startIdx; i<endidx; i++) {
      sum = A[i] + B[i];
   }
   …
}
```

Pthreads Example

```
int main() {
   …
   for (i=0; i<4; i++) {
      pthread_create(&threads[i],
      NULL, add, &data[i]);
   }

   for (i=0; i<4; i++) {
      pthread_join(threads[i],
      (void**)&result[i]);
   }
   …
}
```

```
void *add(void *threadarg) {
   …
   for (i=startIdx; i<endidx; i++) {
      sum = A[i] + B[i];
   }
   …
}
```

Pthreads Example

```
int main() {
   …
   for (i=0; i<4; i++) {
      pthread_create(&threads[i],
      NULL, add, &data[i]);
   }

   for (i=0; i<4; i++) {
      pthread_join(threads[i],
      (void**)&result[i]);
   }
   …
}
```

```
void *add(void *threadarg) {
   …
   for (i=startIdx; i<endidx; i++) {
      sum = A[i] + B[i];
   }
   …
}
```

Pthreads Example

```
int main() {
   …
   for (i=0; i<4; i++) {
      pthread_create(&threads[i],
      NULL, add, &data[i]);
   }

   for (i=0; i<4; i++) {
      pthread_join(threads[i],
      (void**)&result[i]);
   }
   …
}
```

```
void *add(void *threadarg) {
   …
   for (i=startIdx; i<endidx; i++) {
      sum = A[i] + B[i];
   }
   …
}
```

Pthreads Example

```
int main() {
   …
   for (i=0; i<4; i++) {
      pthread_create(&threads[i],
      NULL, add, &data[i]);
   }

   for (i=0; i<4; i++) {
      pthread_join(threads[i],
      (void**)&result[i]);
   }
   …
}
```

```
void *add(void *threadarg) {
   …
   for (i=startIdx; i<endidx; i++) {
      sum = A[i] + B[i];
   }
   …
}
```

Pthreads Example

```
int main() {
    …
    for (i=0; i<4; i++) {
        pthread_create(&threads[i],
        NULL, add, &data[i]);
    }

    for (i=0; i<4; i++) {
        pthread_join(threads[i],
        (void**)&result[i]);
    }
    …
}
```

```
void *add(void *threadarg) {
    …
    for (i=startIdx; i<endidx; i++) {
        sum = A[i] + B[i];
    }
    …
}
```

Pthreads Example

```
int main() {
   …
   for (i=0; i<4; i++) {
      pthread_create(&threads[i],
      NULL, add, &data[i]);
   }

   for (i=0; i<4; i++) {
      pthread_join(threads[i],
      (void**)&result[i]);
   }
   …
}
```

```
void *add(void *threadarg) {
   …
   for (i=startIdx; i<endidx; i++) {
      sum = A[i] + B[i];
   }
   …
}
```

Pthreads Example

```
int main() {
   …
   for (i=0; i<4; i++) {
      pthread_create(&threads[i],
      NULL, add, &data[i]);
   }

   for (i=0; i<4; i++) {
      pthread_join(threads[i],
      (void**)&result[i]);
   }
   …
}
```

```
void *add(void *threadarg) {
   …
   for (i=startIdx; i<endidx; i++) {
      sum = A[i] + B[i];
   }
   …
}
```

Pthreads Example

```
int main() {
   …
   for (i=0; i<4; i++) {
      pthread_create(&threads[i],
      NULL, add, &data[i]);
   }

   for (i=0; i<4; i++) {
      pthread_join(threads[i],
      (void**)&result[i]);
   }
   …
}
```

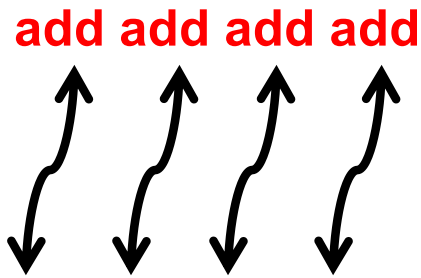
```
void *add(void *threadarg) {
   …
   for (i=startIdx; i<endidx; i++) {
      sum = A[i] + B[i];
   }
   …
}
```

Pthreads Example

```
int main() {
   …
   for (i=0; i<4; i++) {
      pthread_create(&threads[i],
      NULL, add, &data[i]);
   }

   for (i=0; i<4; i++) {
      pthread_join(threads[i],
      (void**)&result[i]);
   }
   …
}
```

```
void *add(void *threadarg) {
   …
   for (i=startIdx; i<endidx; i++) {
      sum = A[i] + B[i];
   }
   …
}
```

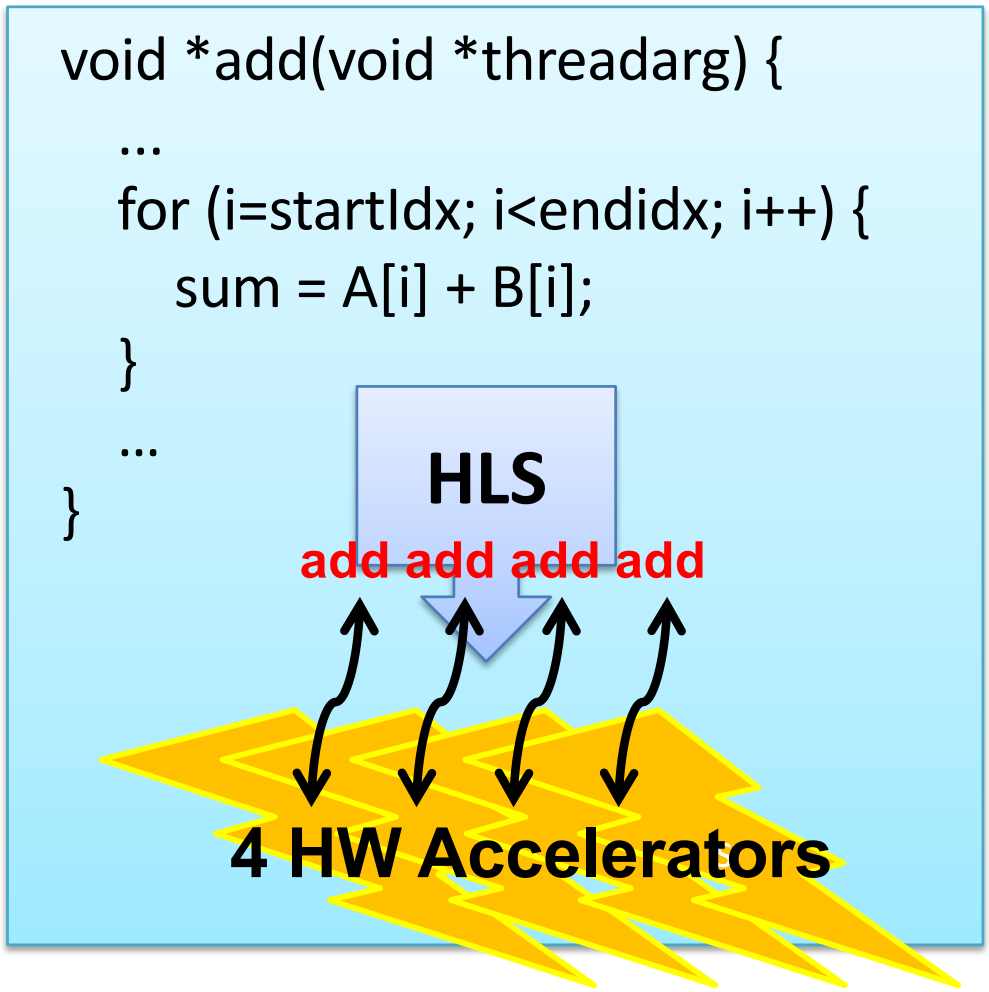


Pthreads in LegUp

```
int main() {
    …
    for (i=0; i<4; i++) {
        pthread_create(&threads[i],
        NULL, add, &data[i]);
    }

    for (i=0; i<4; i++) {
        pthread_join(threads[i],
        (void**)&result[i]);
    }
    …
}
```

```
void *add(void *threadarg) {
    …
    for (i=startIdx; i<endidx; i++) {
        sum = A[i] + B[i];
    }
    …
}
```



Pthreads vs OpenMP

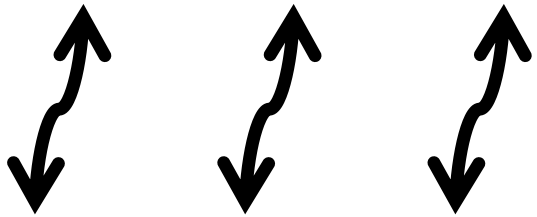
- OpenMP provides an easy/implicit way for parallelizing a section of code (e.g. loops)
- Pthreads require explicit thread forks/joins
- Pthreads can be more work but gives more control to programmer
- Pthreads can execute different functions in parallel

OpenMP/Pthreads Support in LegUp

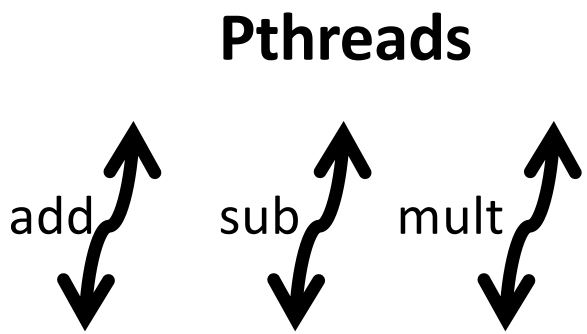
- Allow Pthreads and OpenMP to be used to specify parallel hardware.
- Automatically infer parallel-operating accelerators for the parallel-operating threads.
- Permits a easy exploration of a broad parallelization landscape.
  - Incl. support for *nested parallelism.*

Nested Parallelism

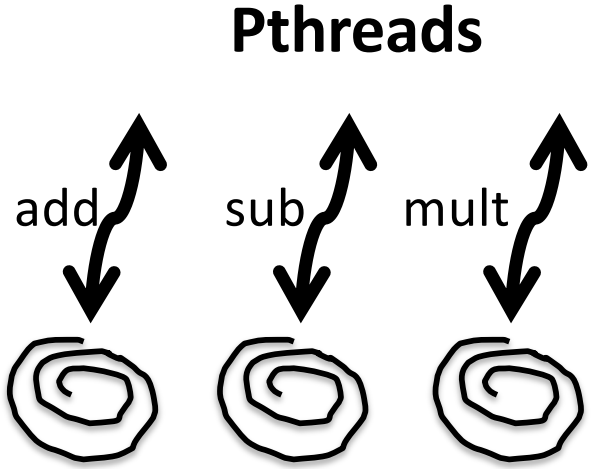
Pthreads



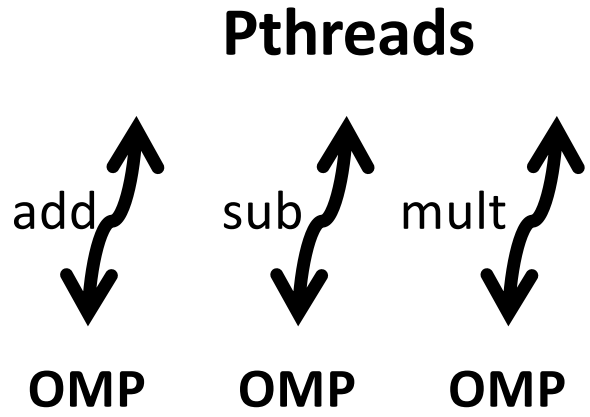
Nested Parallelism



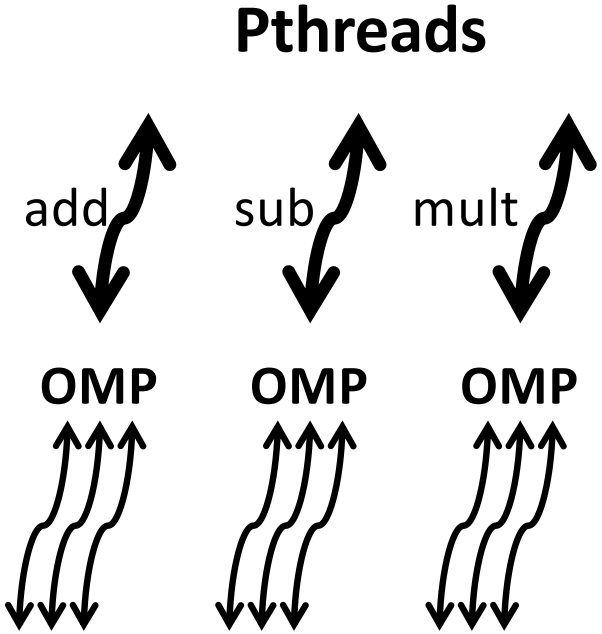
Nested Parallelism



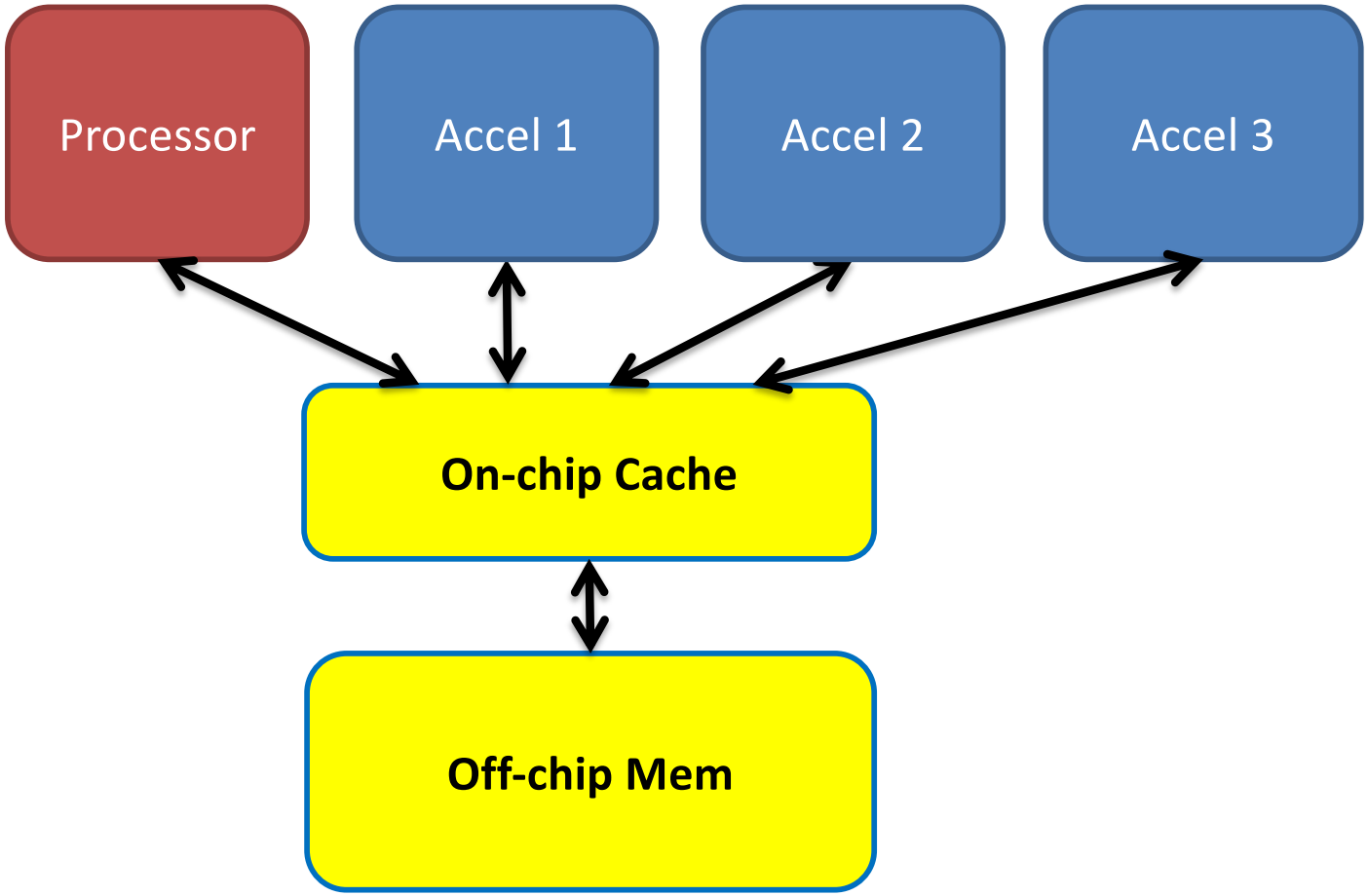
Nested Parallelism



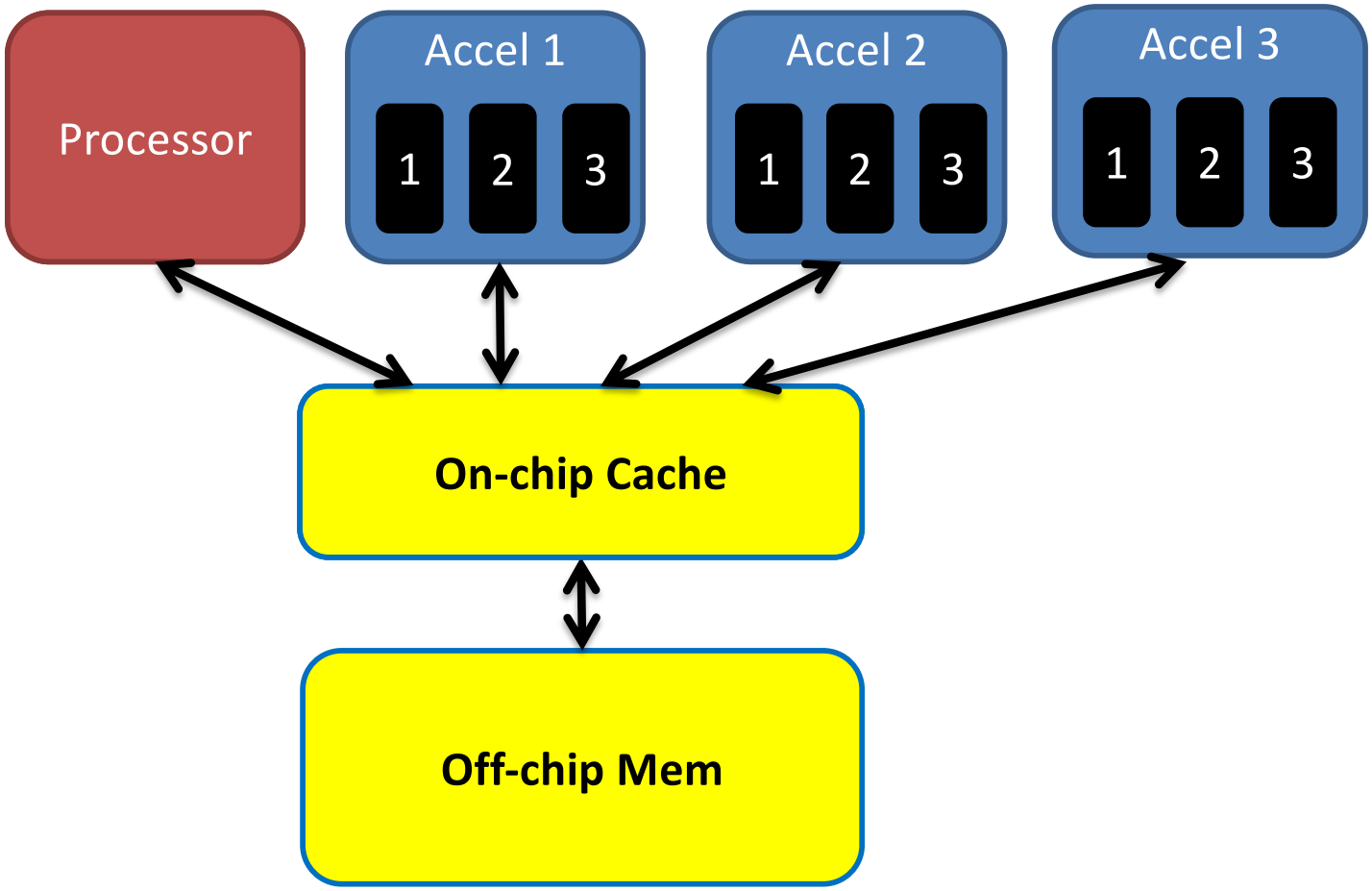
Nested Parallelism



Nested Parallelism



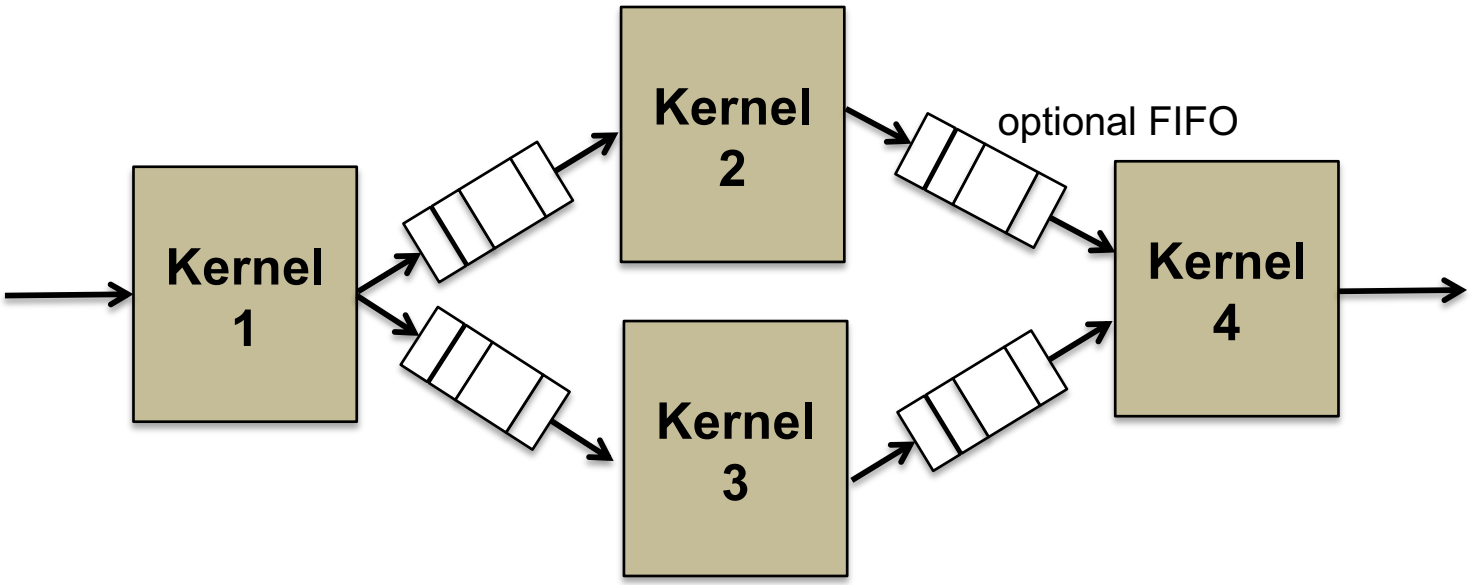
Nested Parallelism



Streaming (Dataflow)

Streaming (Dataflow) Benchmarks

- Interconnected computational “kernels” with optional intermediate FIFOs
- A given kernel can accept new inputs every II cycles (II called initiation interval)
- Kernel may have multiple “tokens” in flight
- Streaming leveraged frequency in audio, video processing, machine learning



Specifying a Streaming Function

- To generate a streaming interface you must specify the function in the LegUp tcl configuration file:

```
function_pipeline FIRFilterStreaming -ii 1
```

- The **initiation interval (II)** is the number of cycles between successive inputs to the streaming kernel
- The user can specify the initiation interval
- II=1 is best but may not be possible due to dependencies and resource usage

Manual Transformations to C code

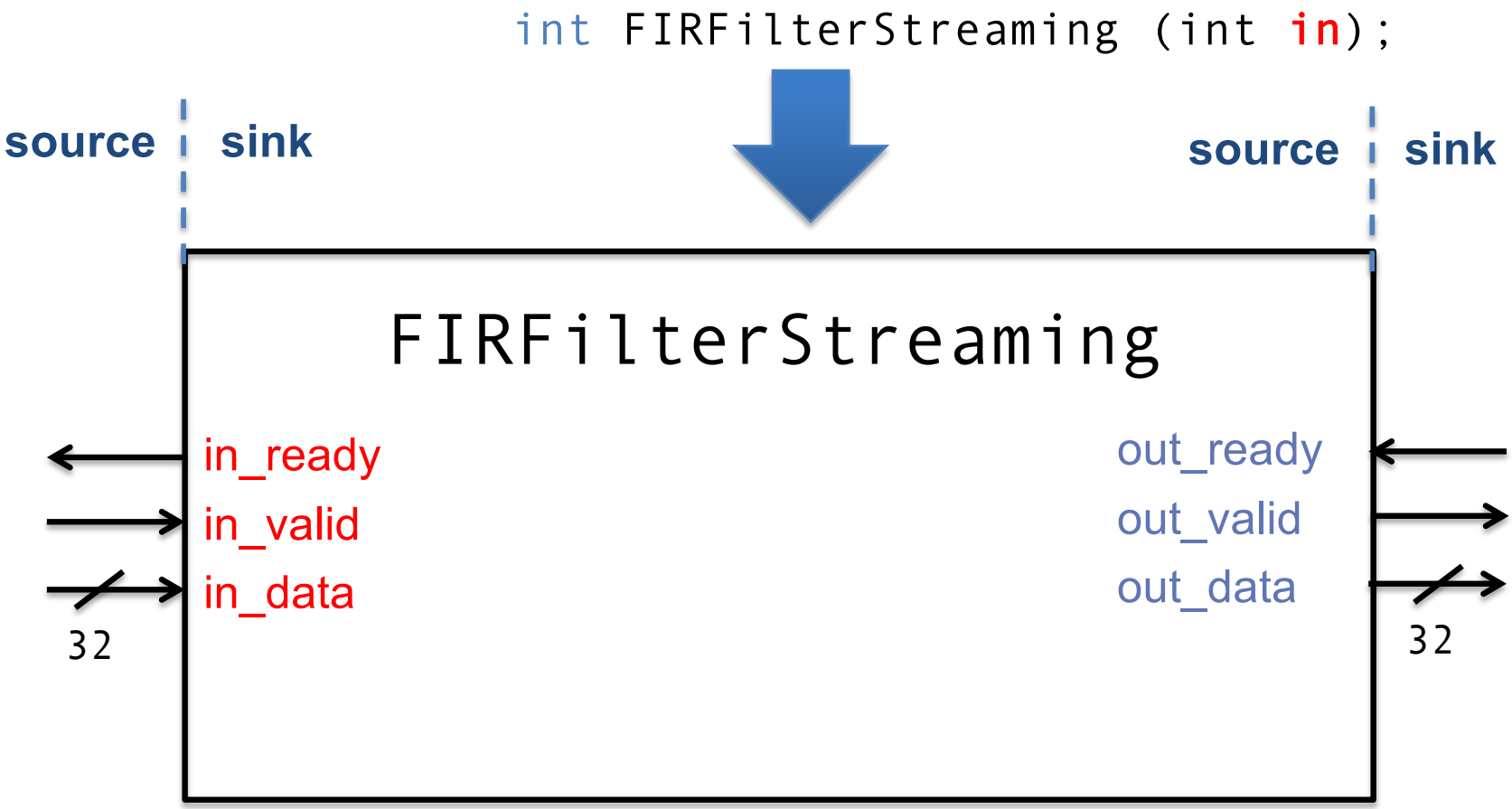
- Changes are likely needed to the original C code
- Streaming function should be passed a LegUp FIFO datastructure instead of passing an array:

```
void gaussian_filter(FIFO *input_fifo,
                     FIFO *output_fifo) {
   int input = fifo_read(input_fifo);
```

- Reduce memory accesses using **static** arrays in function
- Restructure loops

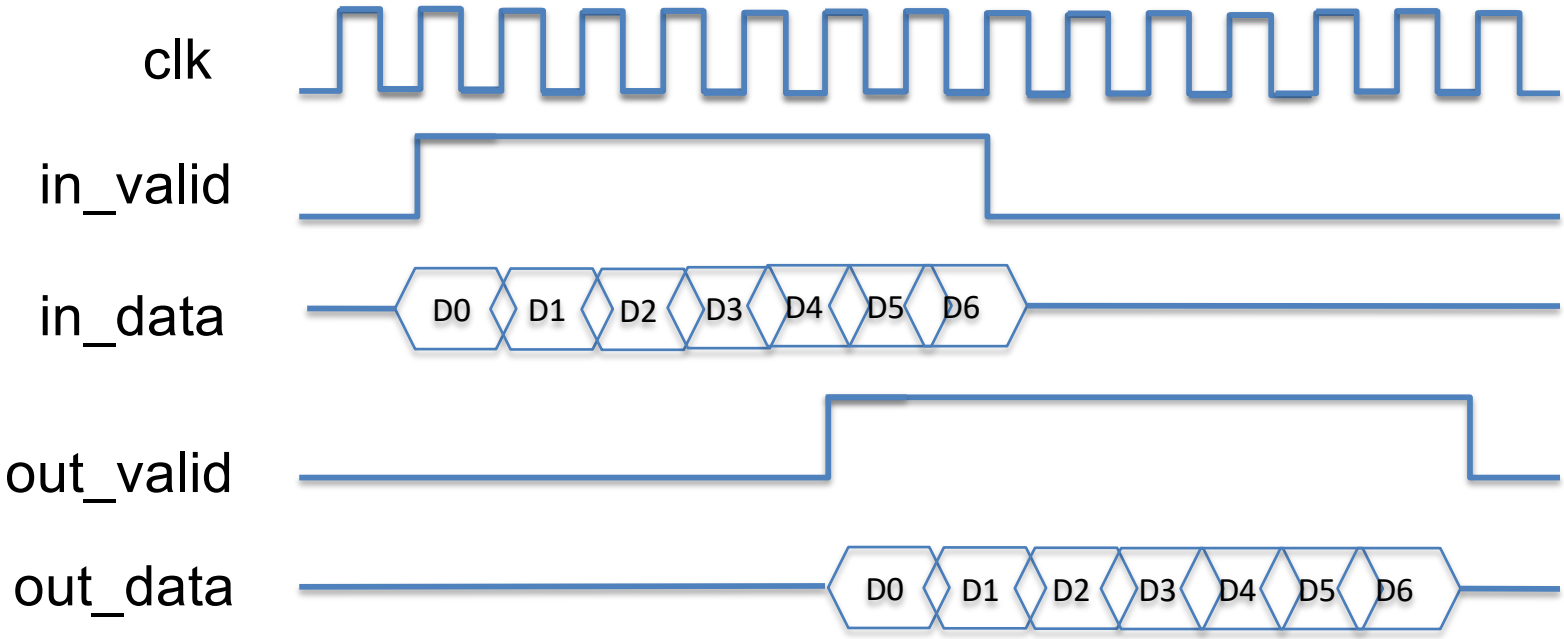
FIR Filter

- 16-tap FIR filter block diagram
- New input/output every clock cycle. First output after 16 cycles



Streaming summary

- LegUp-generated block (ready/valid):



in_ready = out_ready = 1

Streaming Pixel Inputs

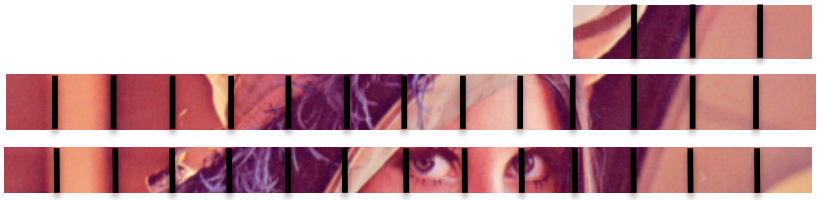
- New pixel arrives every cycle
- We would like to perform convolution as each pixel arrives

Required Pixel Memory

- Which pixels do we need to store in memory?
- Observe: we only need the previous two rows of pixels

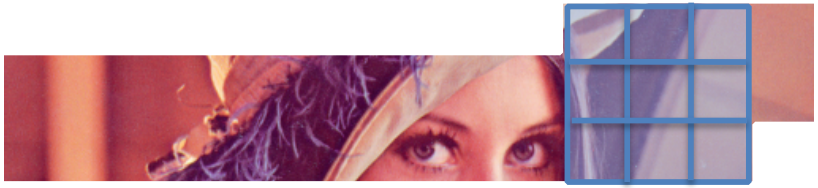
Pixel Shift Register

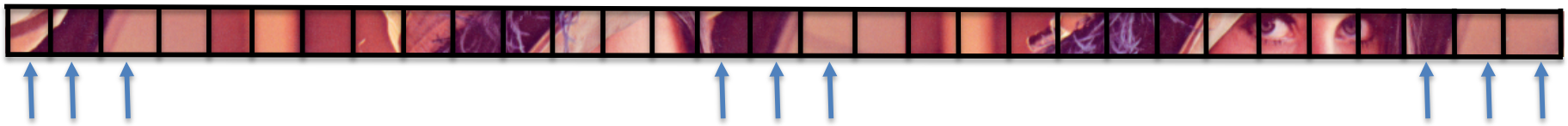
- We store the previous two rows in a shift register
- Efficient hardware implementation



Pixel Shift Register

- Now we shift in every new pixel and discard oldest pixel
- Perform dot product on 9 highlighted pixels

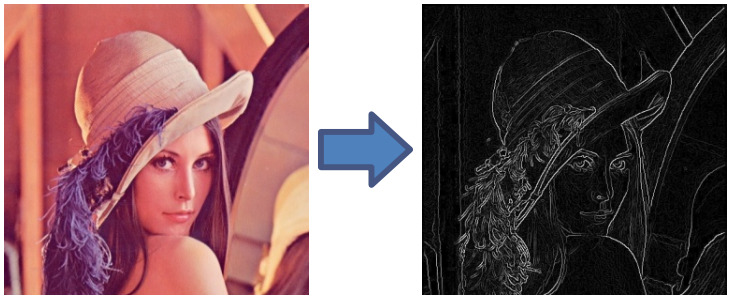




Best Applications for HLS

- Video/image/audio stream processing (algorithmic-heavy)

Image Filtering



Audio Beamforming

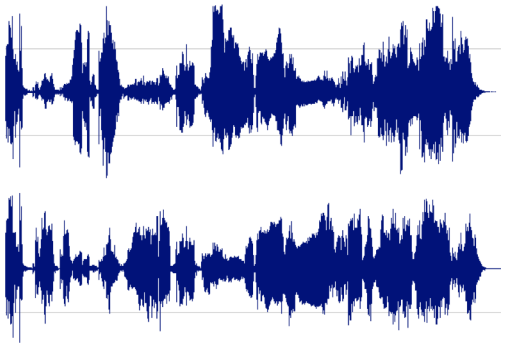
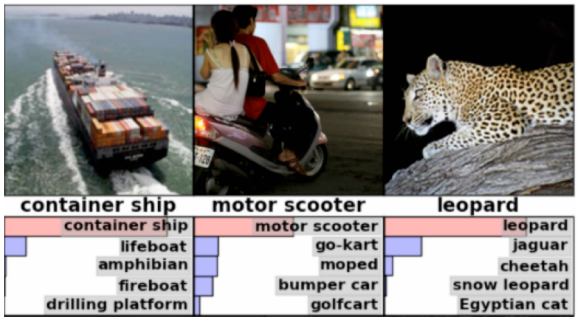
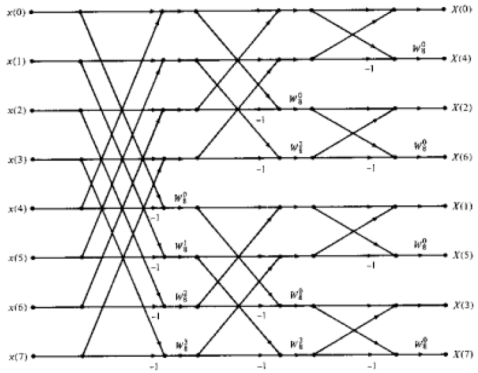


Image Recognition



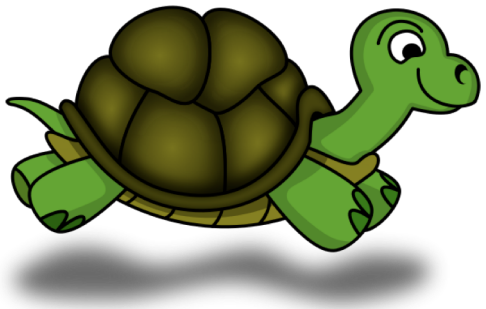
- Avoid using C code for describing:
  - Unique hardware structure: FFT butterfly
  - Cycle-accurate hardware: Bus controller



HLS Research Challenges

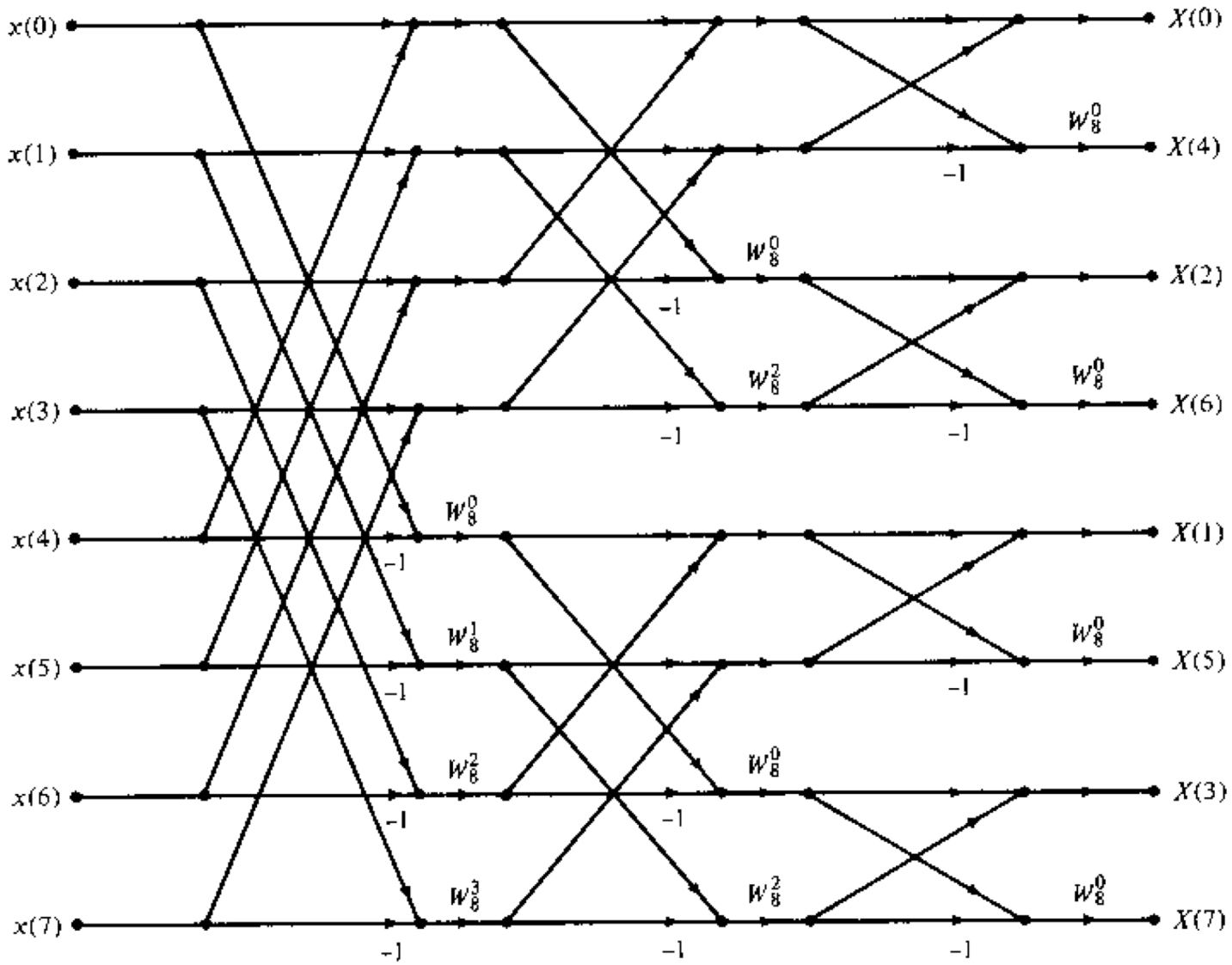
Quality of the Hardware

- Performance of HLS-generated circuits not as good as human-designed circuits



- However, HLS-generated circuits are already better than SW in many cases

FFT: Hard to Auto-Synthesize



Syntactic Variance / Constraints

- HLS tool QoR highly sensitive to style of input code + constraints
- LegUp HLS example:

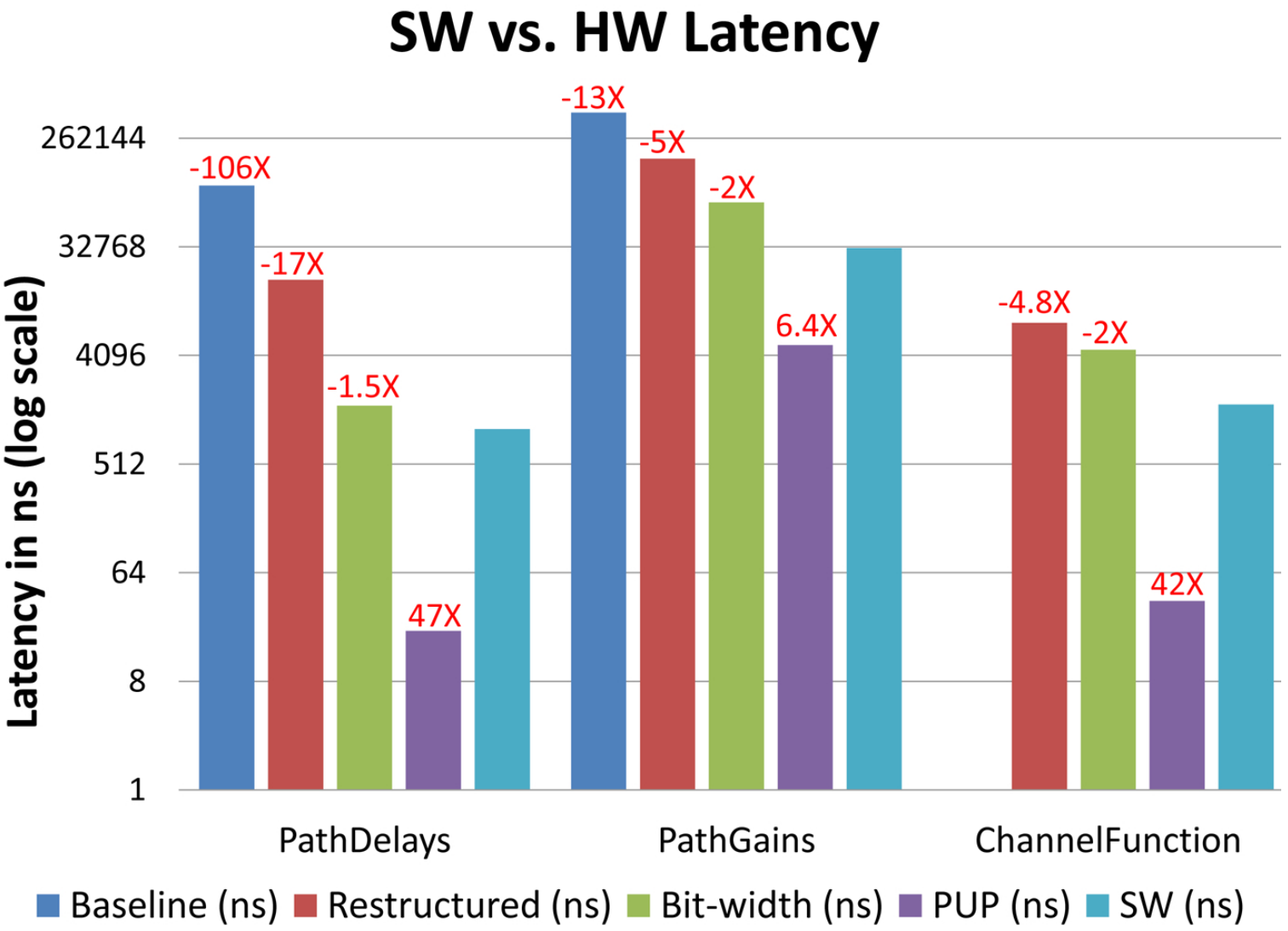
```
for (i = 0; i < 100; i++) {
    if (A[i] & 1)
        sum += A[i];
    else
        sum -= A[i];
}
```

Cannot loop pipeline

```
for (i = 0; i < 100; i++) {
    temp1 = sum + A[i];
    temp2 = sum – A[i];
    sum = (A[i] & 1) ?
                temp1 : temp2;
}
```

Can loop pipeline

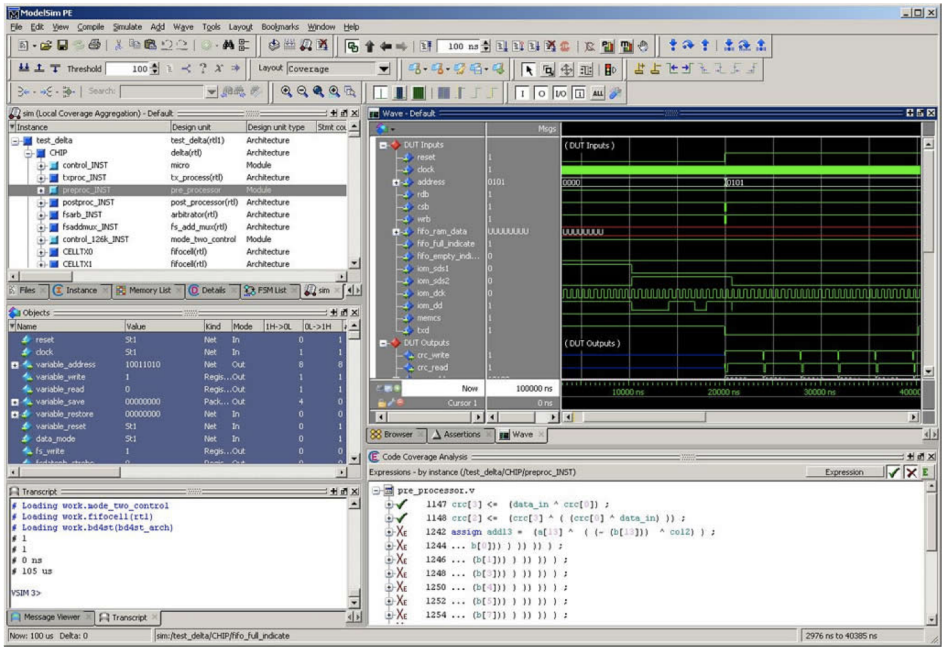
Syntactic Variance / Constraints (2)



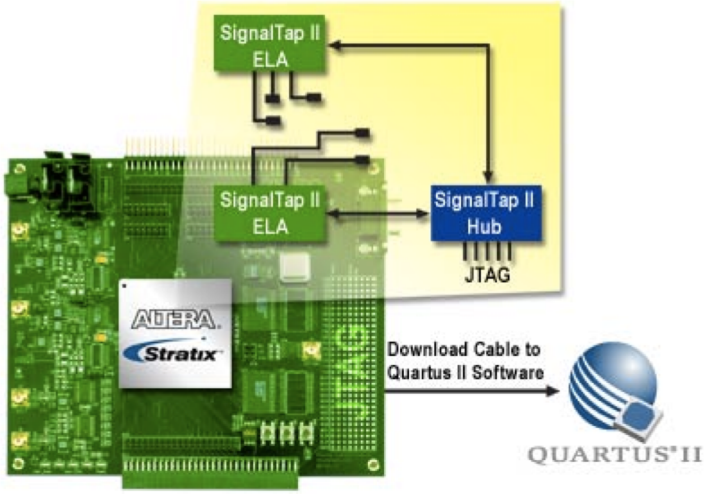
Matai et al., “Designing a Hardware in the Loop Wireless Digital Channel Emulator for Software Defined Radio”, FPT 2012.

Debugging

- Invariably… things go wrong, e.g.:
  - Integration of synthesized HW in system
  - Silicon issues: timing, reliability (SEUs)
- Today's HLS:



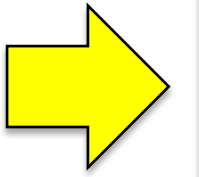
Chip ScopePro

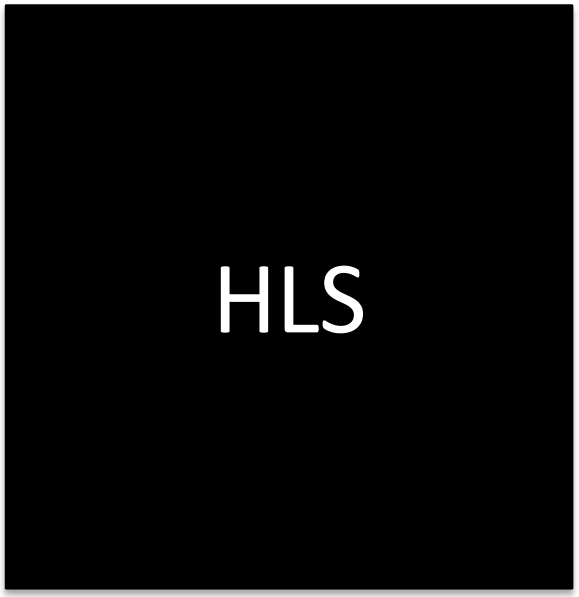


Visualization

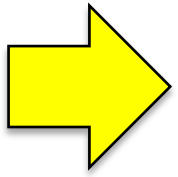
- Today's HLS:

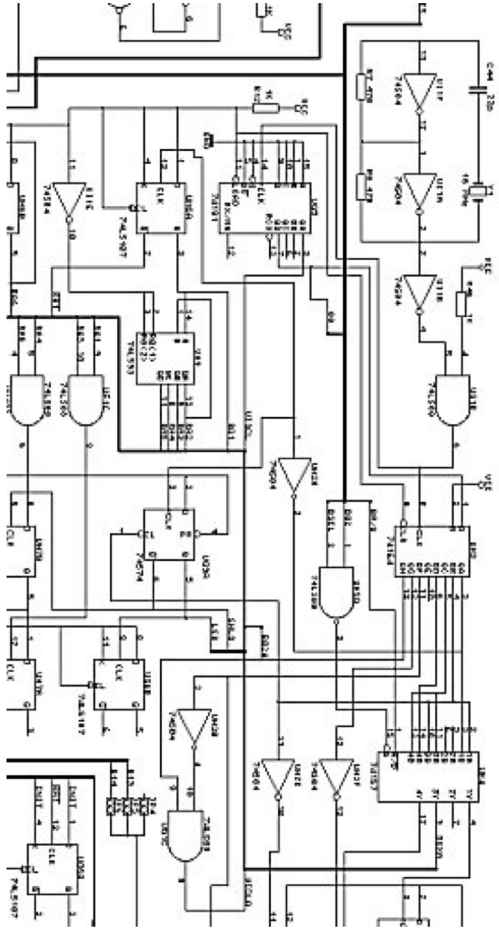
```
for (i = 0; i < 4*Nk; i+
    W[i] = key[i];
for (i = Nk; i < Nb*(Nr+
    {
    temp = Wb[i-1];
    if ((i%Nk) == 0)
        temp = SubByte(R
    Wb[i] = Wb[i - Nk]^t
    }
```





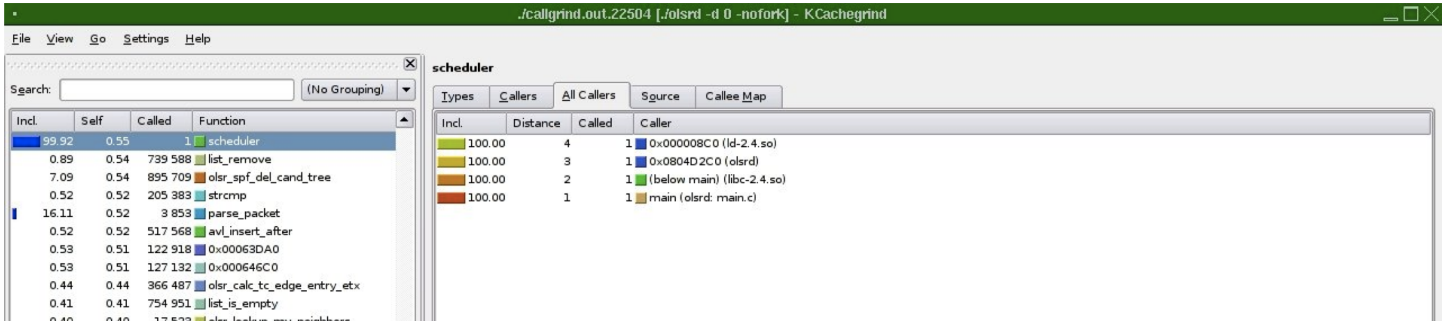
"Black box"



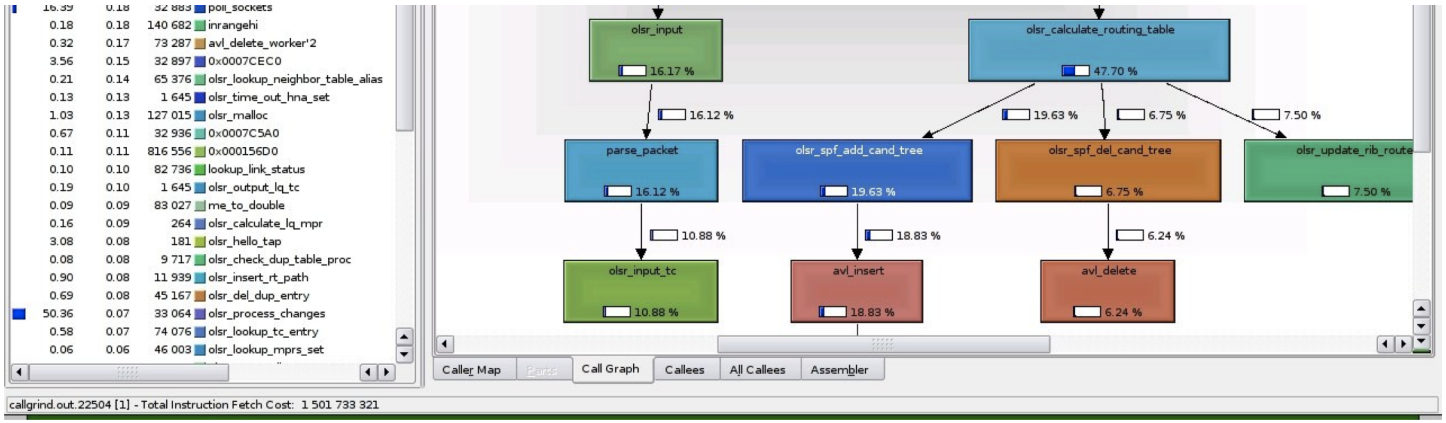


(hundreds/tens) thousands of lines of HDL code

Visualization (2)



"SW-engineer comprehensible"
HW visualization capabilities are needed that guide HW optimization



Common Benchmarking

- No accepted benchmark suite for HLS
  - CHStone circuits don't stress capabilities of modern tools
- No accepted benchmarking methodology:
  - Push button?
  - Constraints, pragmas?
- "Insecurity" among HLS commercial vendors
  - Vendors do not permit results to be published

Questions?

http://janders.eecg.utoronto.ca

janders@ece.utoronto.ca