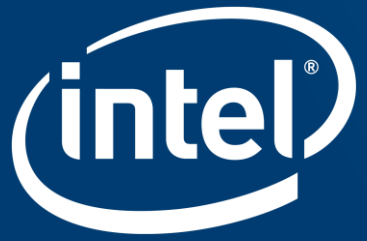
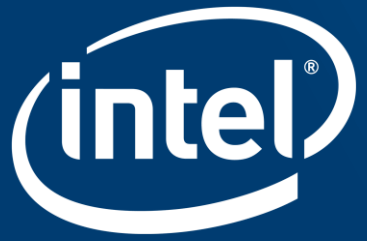


Targeting FPGAs with an LLVM compiler

Dmitry Denisenko

Intel Programmable Solutions Group

November 13, 2016, LLVM-HPC3 SC'16, Salt Lake City, UT



FPGA Overview

FPGAs are Everywhere!

Consumer Automotive

Entertainment
Broadband
Audio/video
Video display

Automotive
Navigation
Entertainment

Test, Measurement, & Medical

Instrumentation
Medical
Test equipment
Manufacturing

Communications Broadcast

Wireless
Cellular
Basestations
Wireless LAN

Networking
Switches
Routers

Wireline
Optical
Metro
Access

Broadcast
Studio
Satellite
Broadcasting

Military & Industrial

Military
Secure comm.
Radar
Guidance and control

Security & Energy Management
Card readers
Control systems
ATM

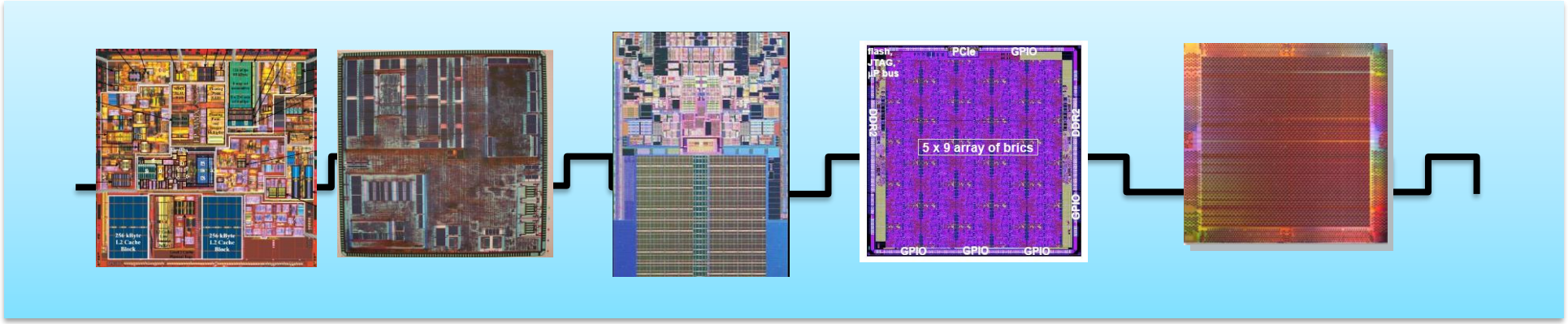
Computer & Storage

Computers
Servers
Mainframe

Storage
RAID
SAN

Office Automation
Copiers
Printers
MFP

Spectrum of approaches to high performance



CPUs | DSPs | Multi-Cores | Arrays | FPGAs

Single Cores

Multi-Cores
Coarse-Grained
CPUs and DSPs

Coarse-Grained
Massively
Parallel
Processor
Arrays

Fine-Grained
Massively
Parallel
Arrays

What's in my FPGA?

DSPs

- Dedicated single-precision floating point multiply and accumulators

Block RAMs

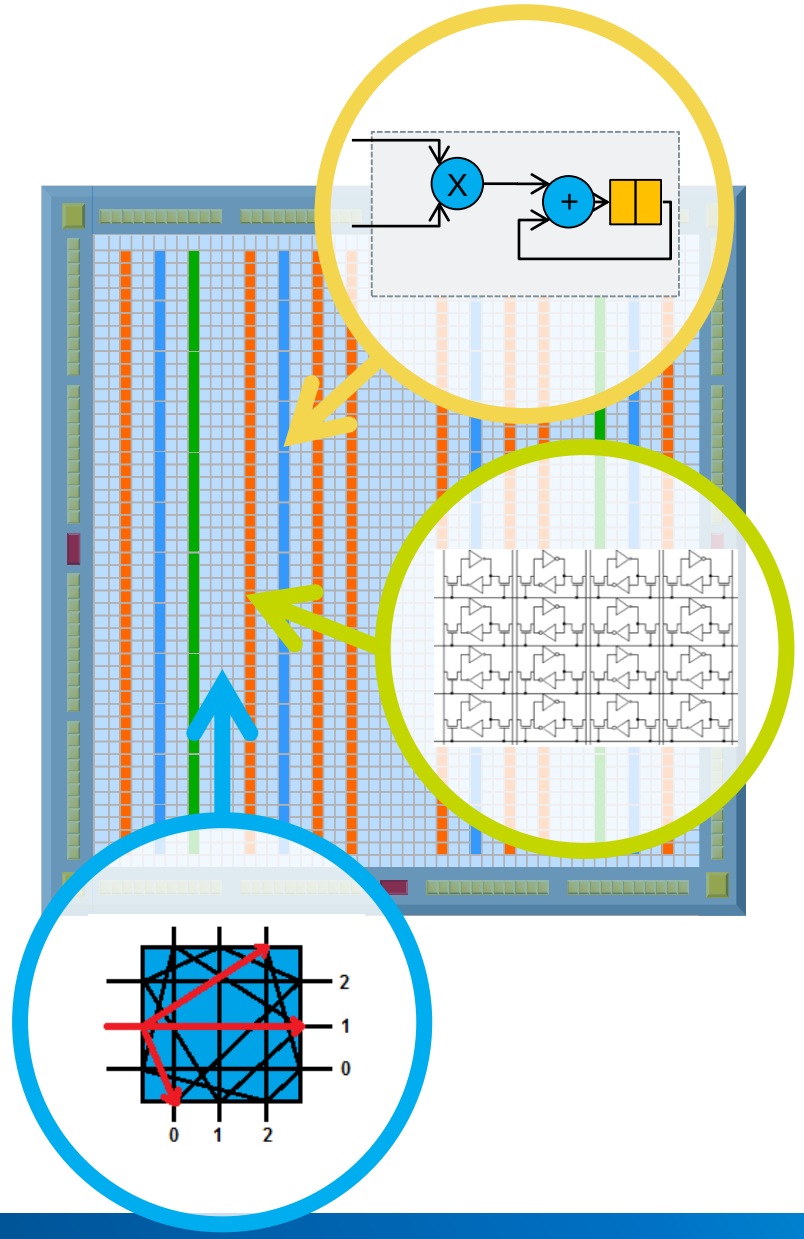
- Small embedded memories that can be stitched to form an arbitrary memory system

Arithmetic Logic Modules

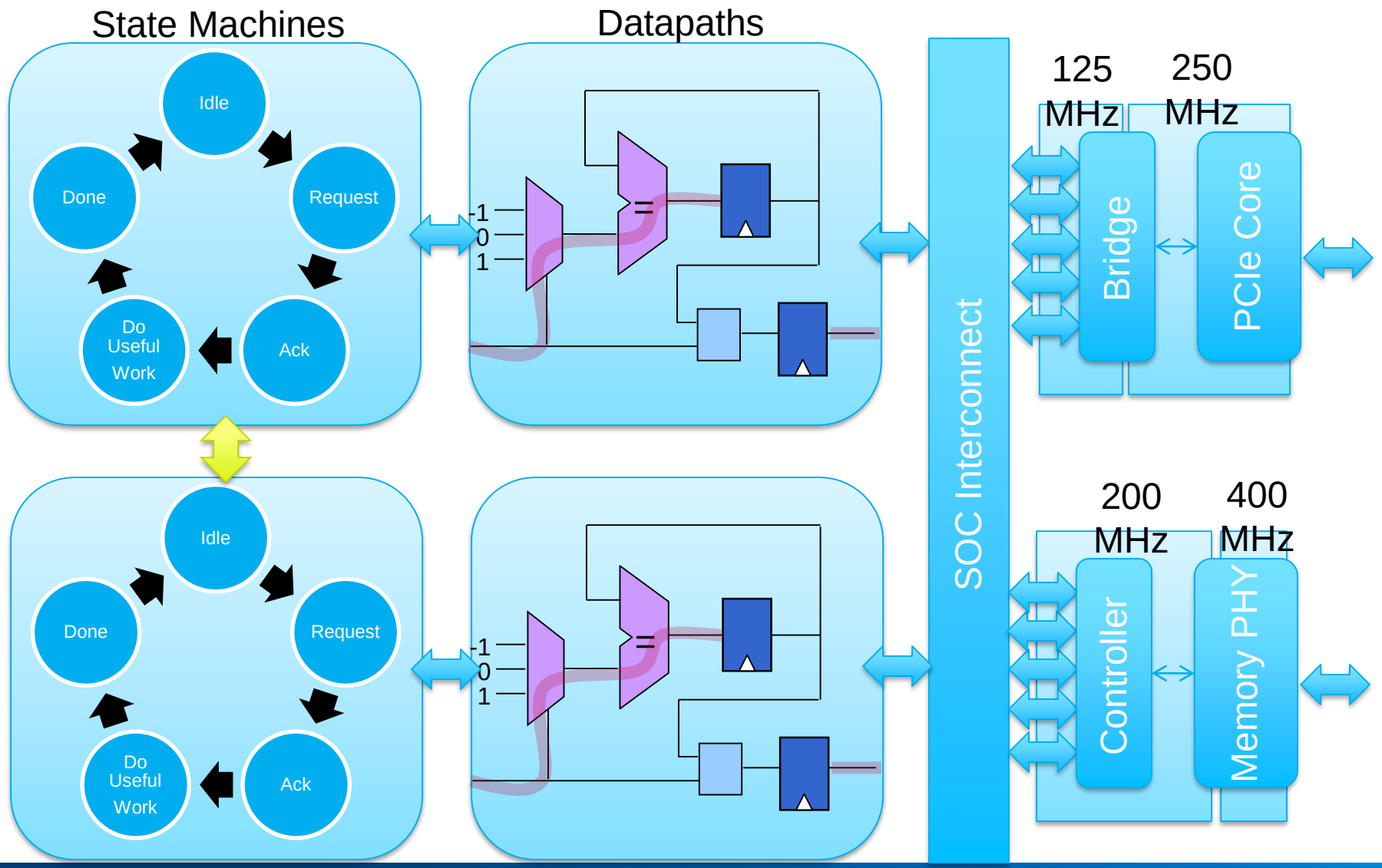
- Implement arbitrary logic functions

Programmable Interconnect

- Programmable routing that can build arbitrary topologies



FPGA Hardware Design



Hardware Design Entry Complexity

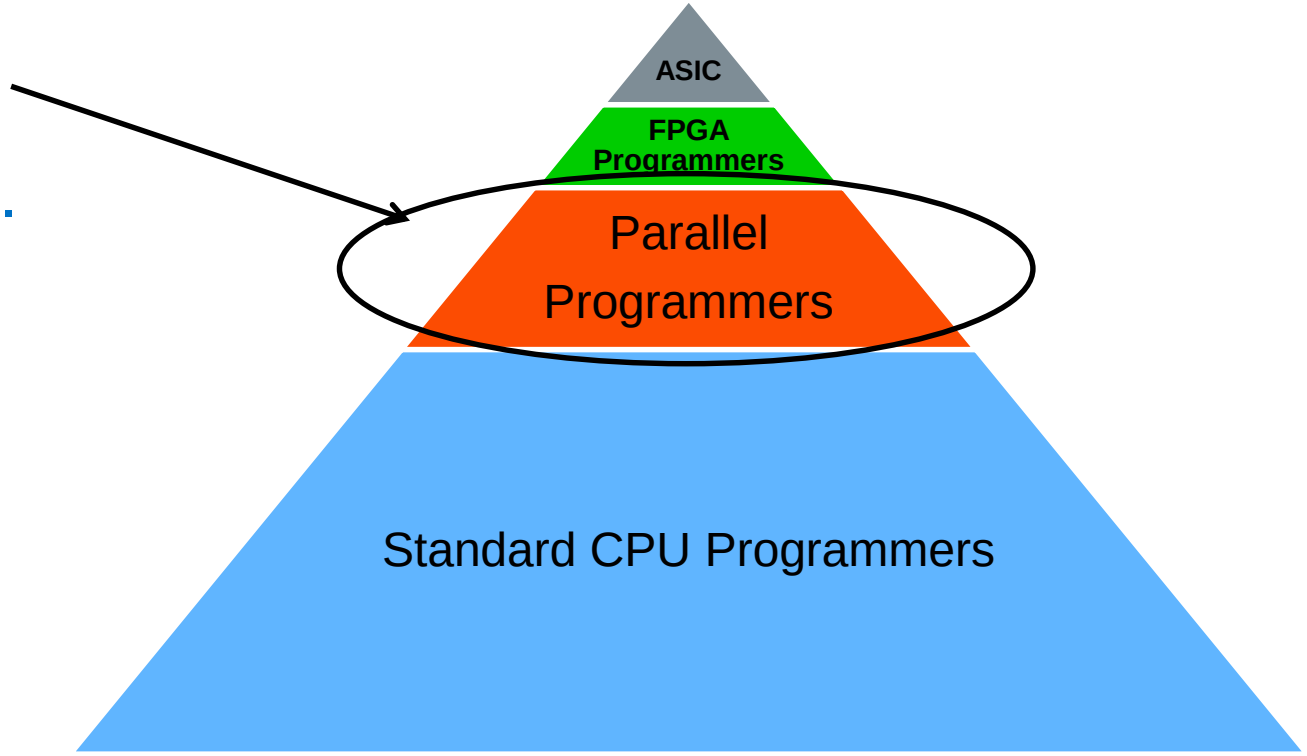
Traditional description of these circuits is done through **Hardware Design Languages** such as VHDL or Verilog.

Incredibly detailed design must be done before a first working version is possible

- Cycle by cycle behavior must be specified for every register in the design
- The complete flexibility of the FPGA means that the designer needs to specify all aspects of the hardware circuit
  - Buffering, Arbitration, IP Core interfacing, etc

Why OpenCL on FPGAs

Intel FPGA SDK for OpenCL is an LLVM-based compiler that raises the level of abstraction for FPGA design to make it accessible to more people.



FPGAs vs CPUs

FPGAs are dramatically different than CPUs

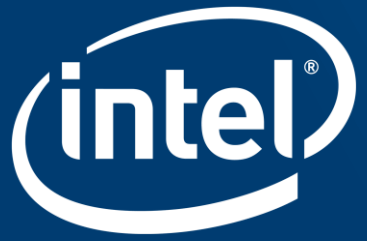
- Massive fine-grained parallelism
- Complete configurability
- Huge internal bandwidth
- No callstack
- No dynamic memory allocation
- Very different instruction costs
- No fixed number of program registers
- No fixed memory system
- Much more flexibility with data types

Targeting an Architecture

In a CPU, the program is mapped to a fixed architecture

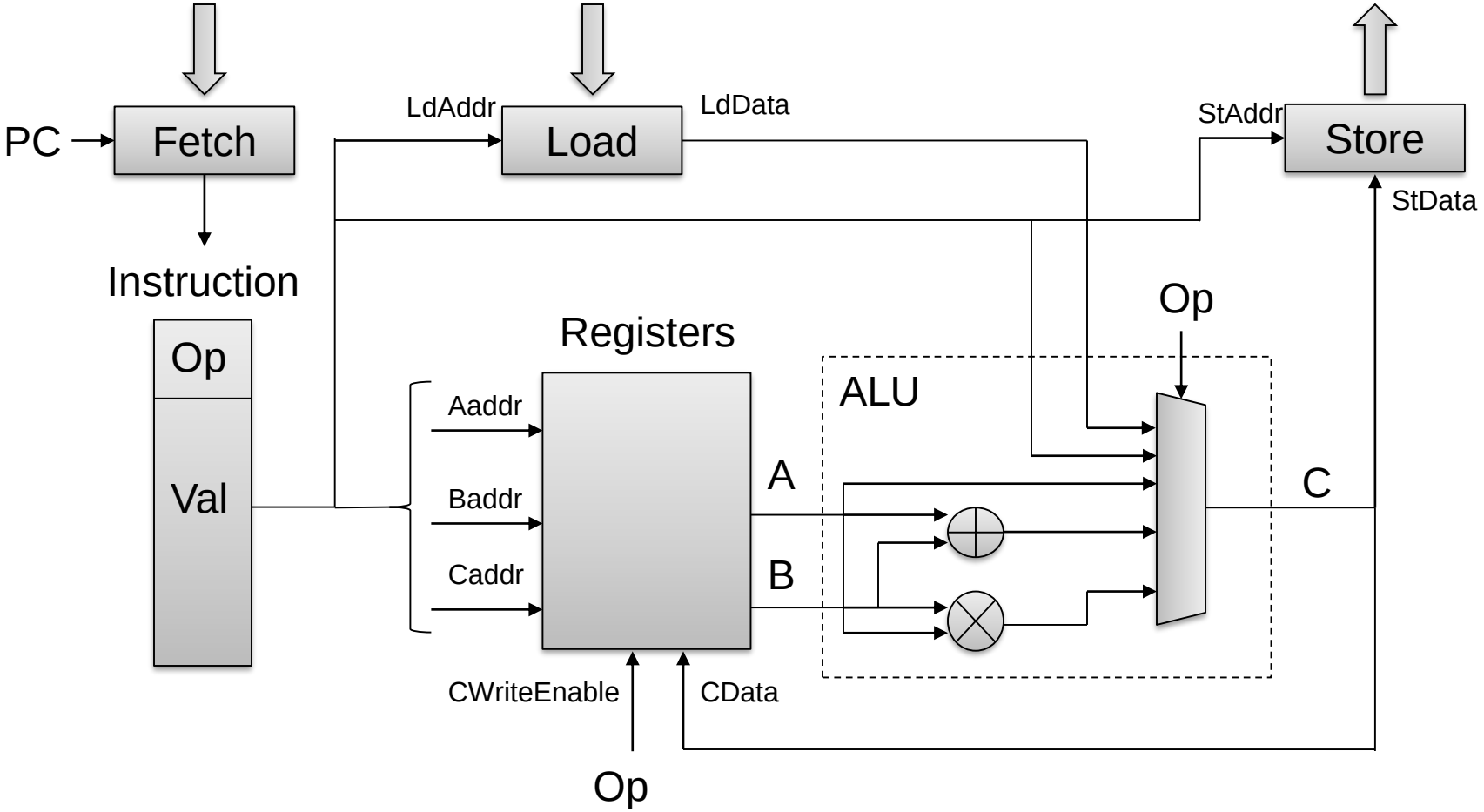
In an FPGA, there is NO fixed architecture

The program defines the architecture

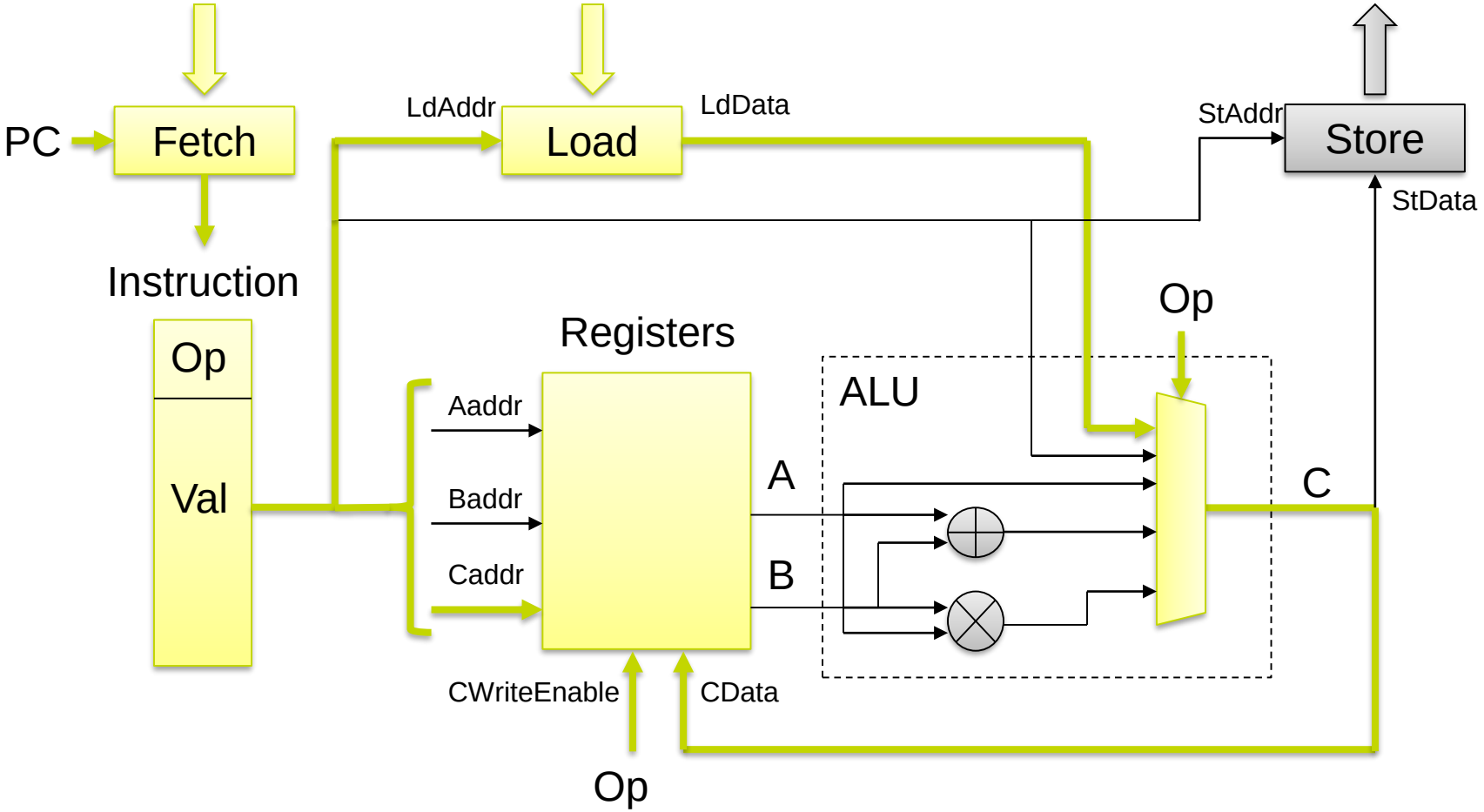


1. Computation in Space

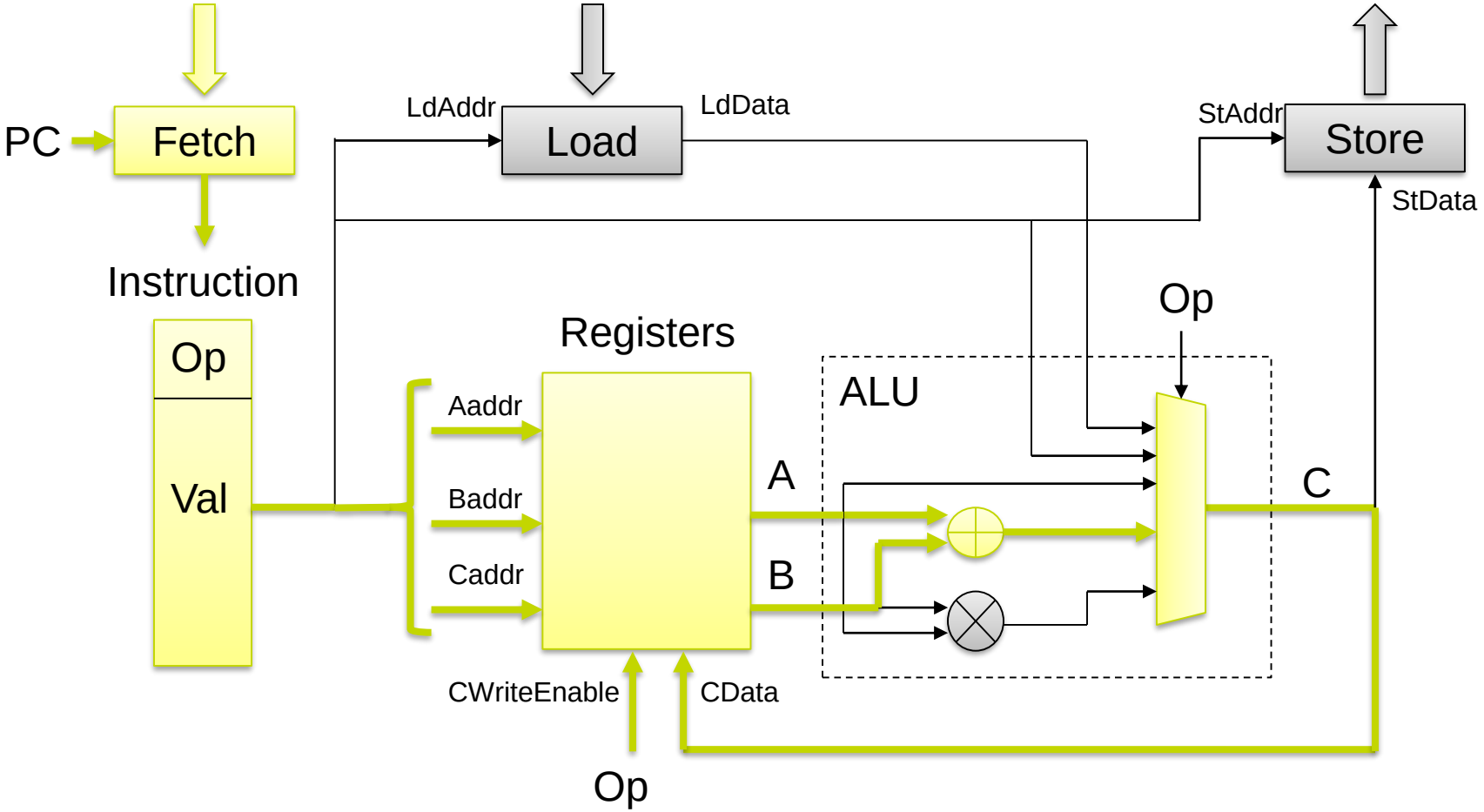
A simple 3-address CPU



Load memory value into register



Add two registers, store result in register



A simple program

Mem[100] += 42 * Mem[101]

CPU instructions:

R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]

CPU activity, step by step

R0 ← Load Mem[100]

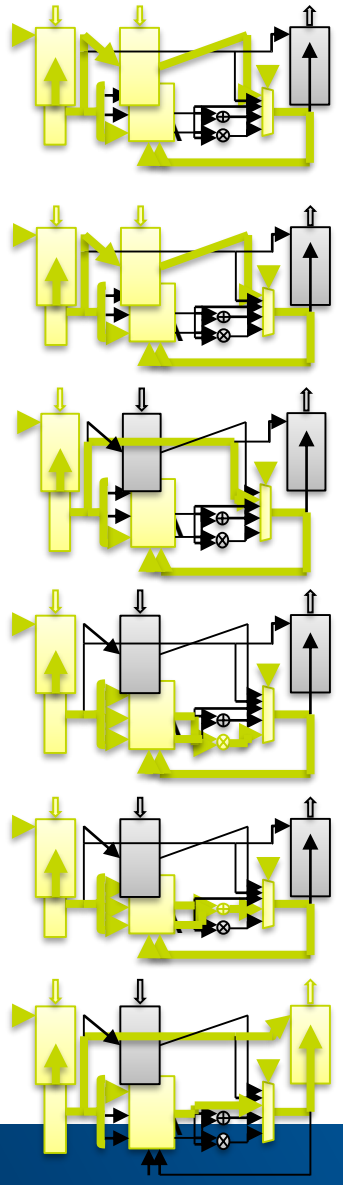
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]



Time

Unroll the CPU hardware…

R0 ← Load Mem[100]

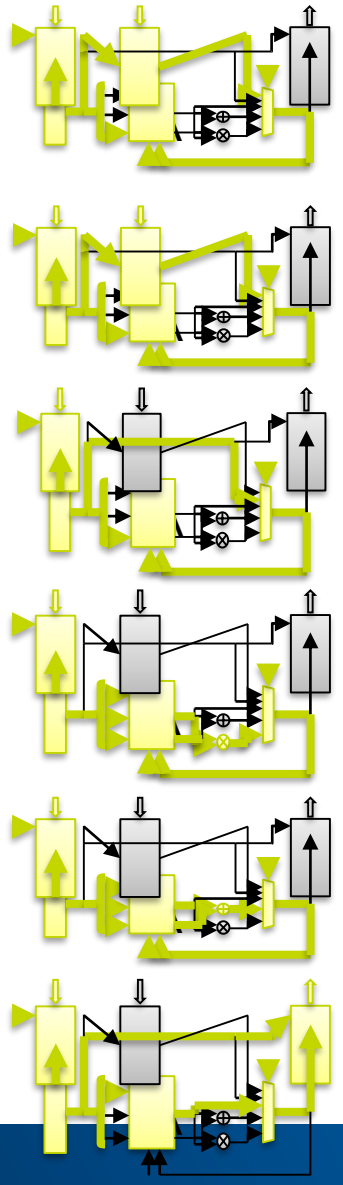
R1 ← Load Mem[101]

R2 ← Load #42

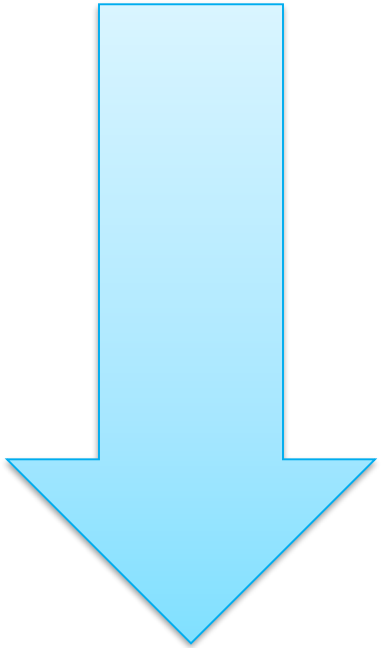
R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]

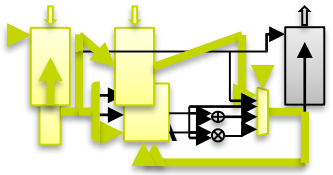


Space

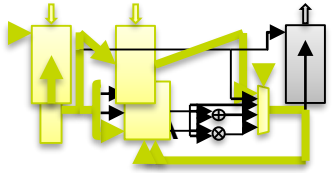


… and specialize by position

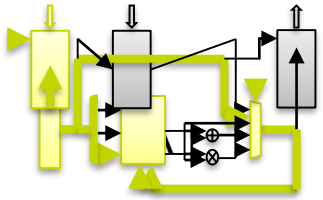
R0 ← Load Mem[100]



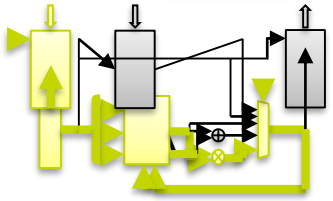
R1 ← Load Mem[101]



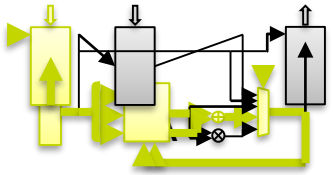
R2 ← Load #42



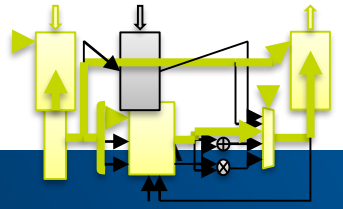
R2 ← Mul R1, R2



R0 ← Add R2, R0



Store R0 → Mem[100]



1. Instructions are fixed. Remove “Fetch”

… and specialize

R0 ← Load Mem[100]

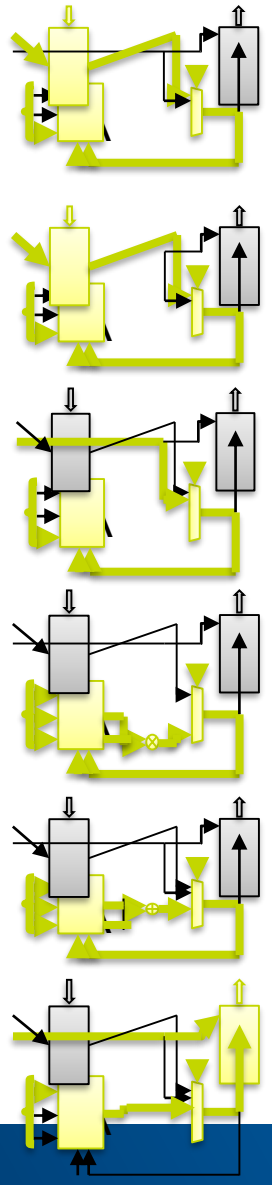
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

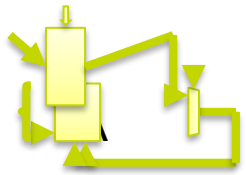
Store R0 → Mem[100]



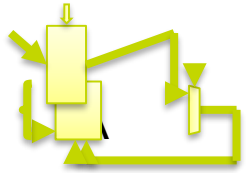
1. Instructions are fixed.
   Remove “Fetch”
2. Remove unused ALU ops

… and specialize

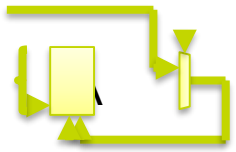
R0 ← Load Mem[100]



R1 ← Load Mem[101]

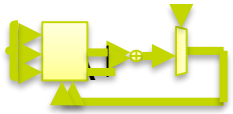


R2 ← Load #42



R2 ← Mul R1, R2

R0 ← Add R2, R0



Store R0 → Mem[100]

1. Instructions are fixed.
   Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store

… and specialize

R0 ← Load Mem[100]

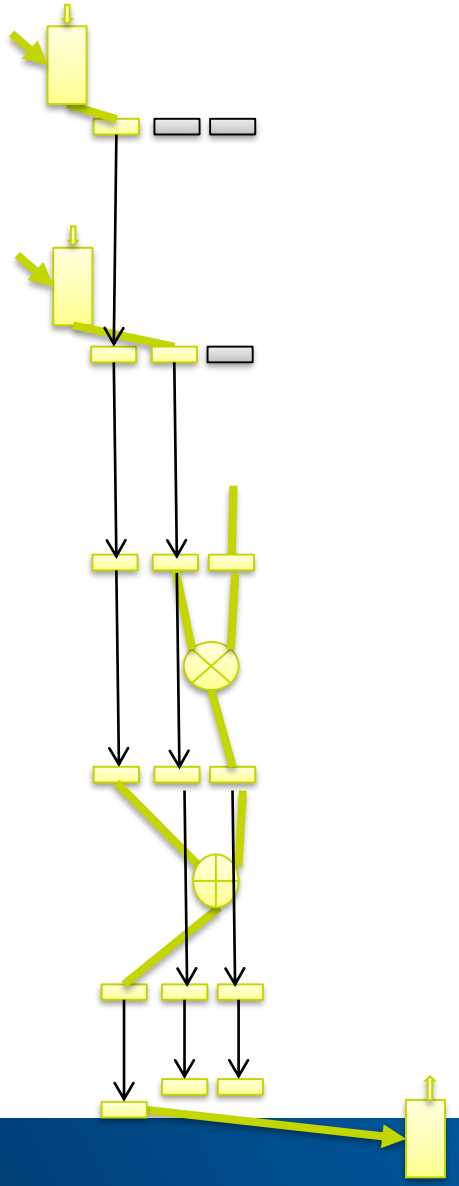
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]



1. Instructions are fixed.
   Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly!
   And propagate state.

… and specialize

R0 ← Load Mem[100]

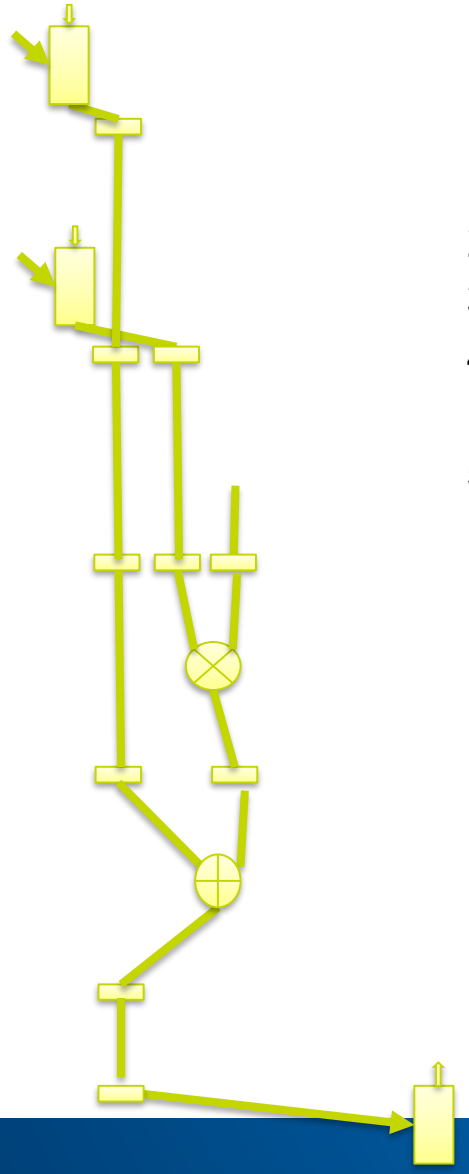
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]



1. Instructions are fixed.
   Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly!
   And propagate state.
5. Remove dead data.

Optimize the Datapath

R0 ← Load Mem[100]

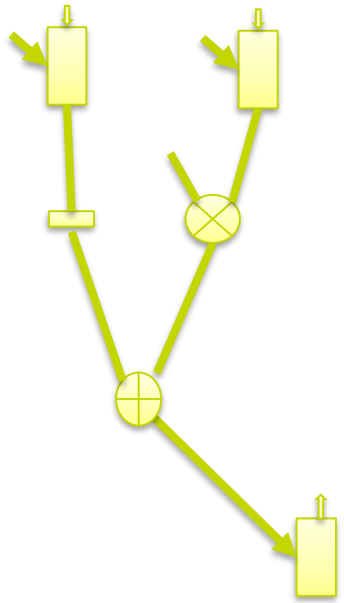
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

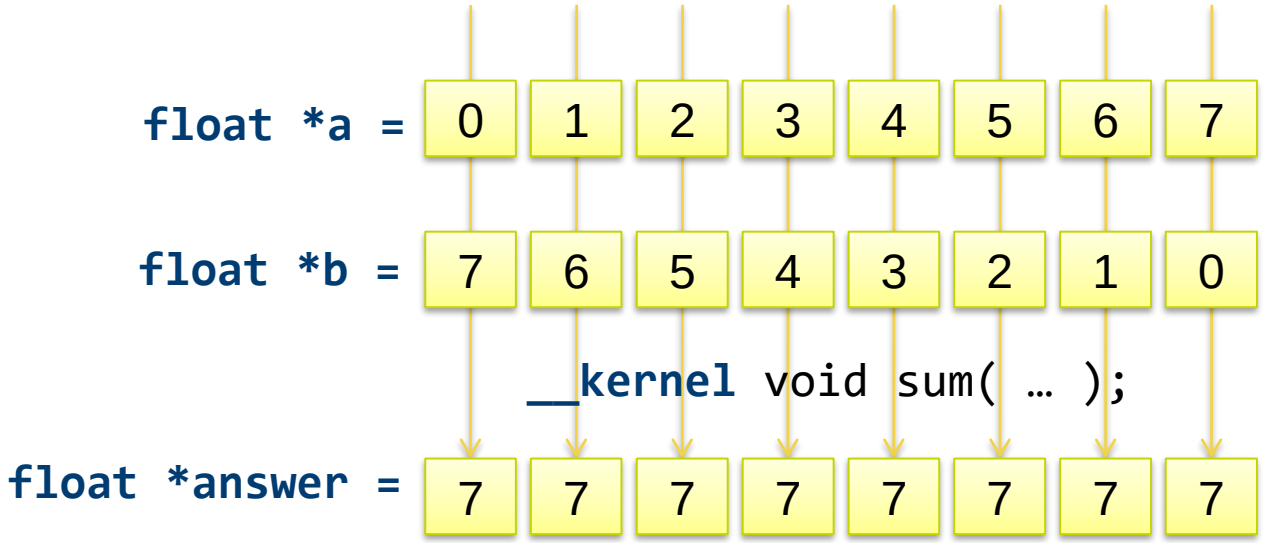
Store R0 → Mem[100]



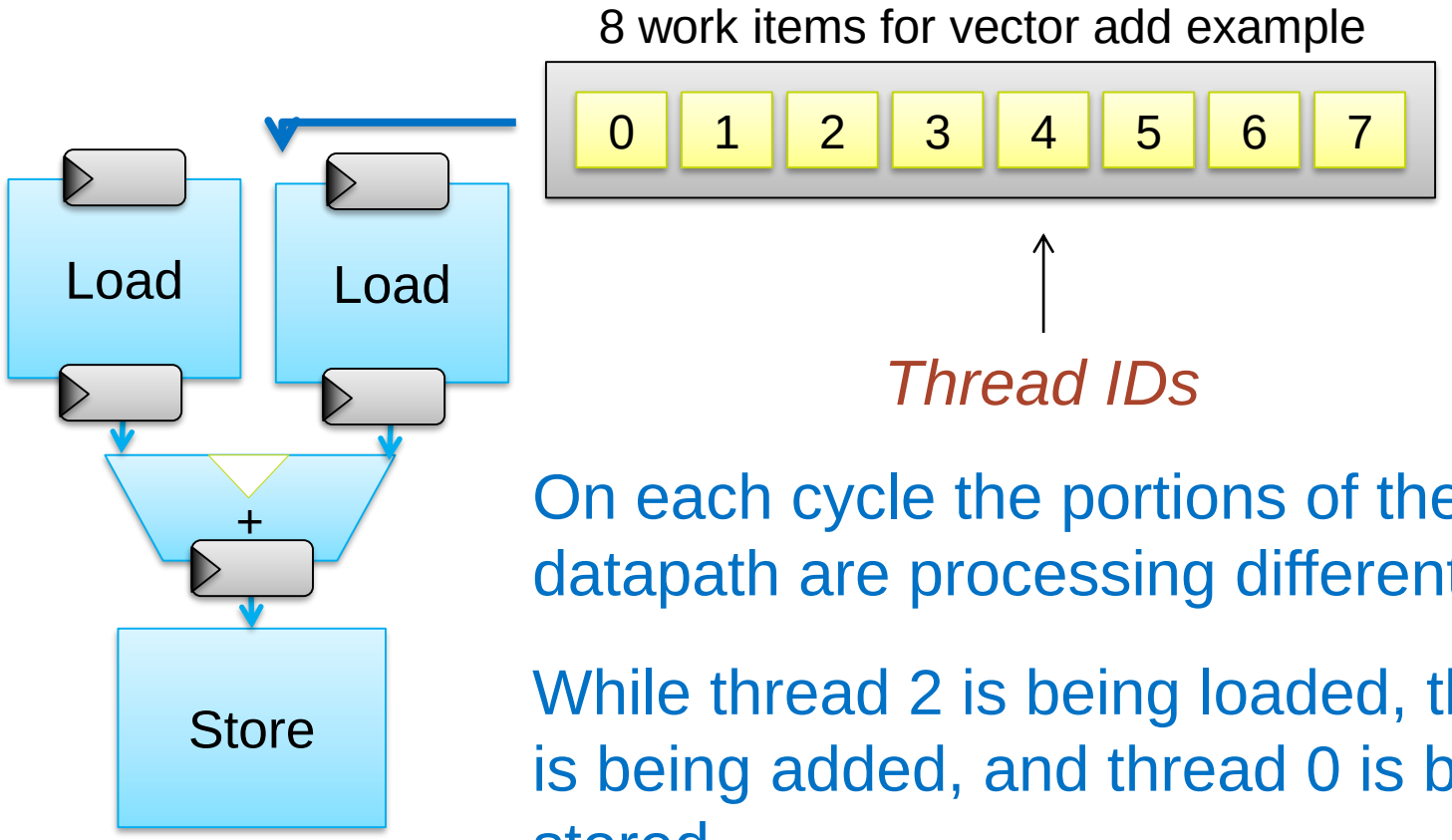
1. Instructions are fixed.
   Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly!
   And propagate state.
5. Remove dead data.
6. Reschedule!

Data parallel kernel

```
__kernel void
sum(__global const float *a,
__global const float *b,
__global float *answer)
{
int xid = get_global_id(0);
answer[xid] = a[xid] + b[xid];
}
```



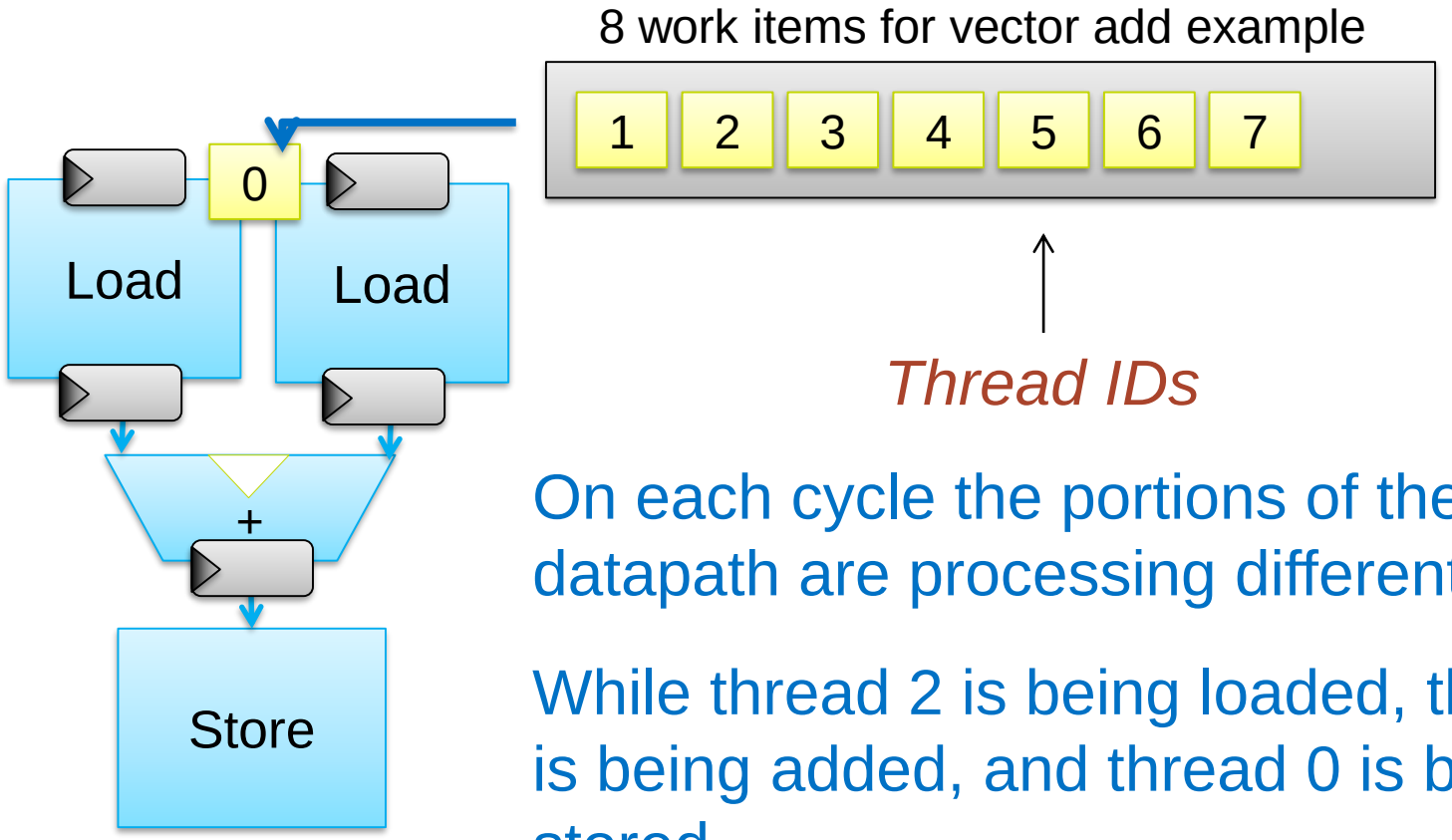
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

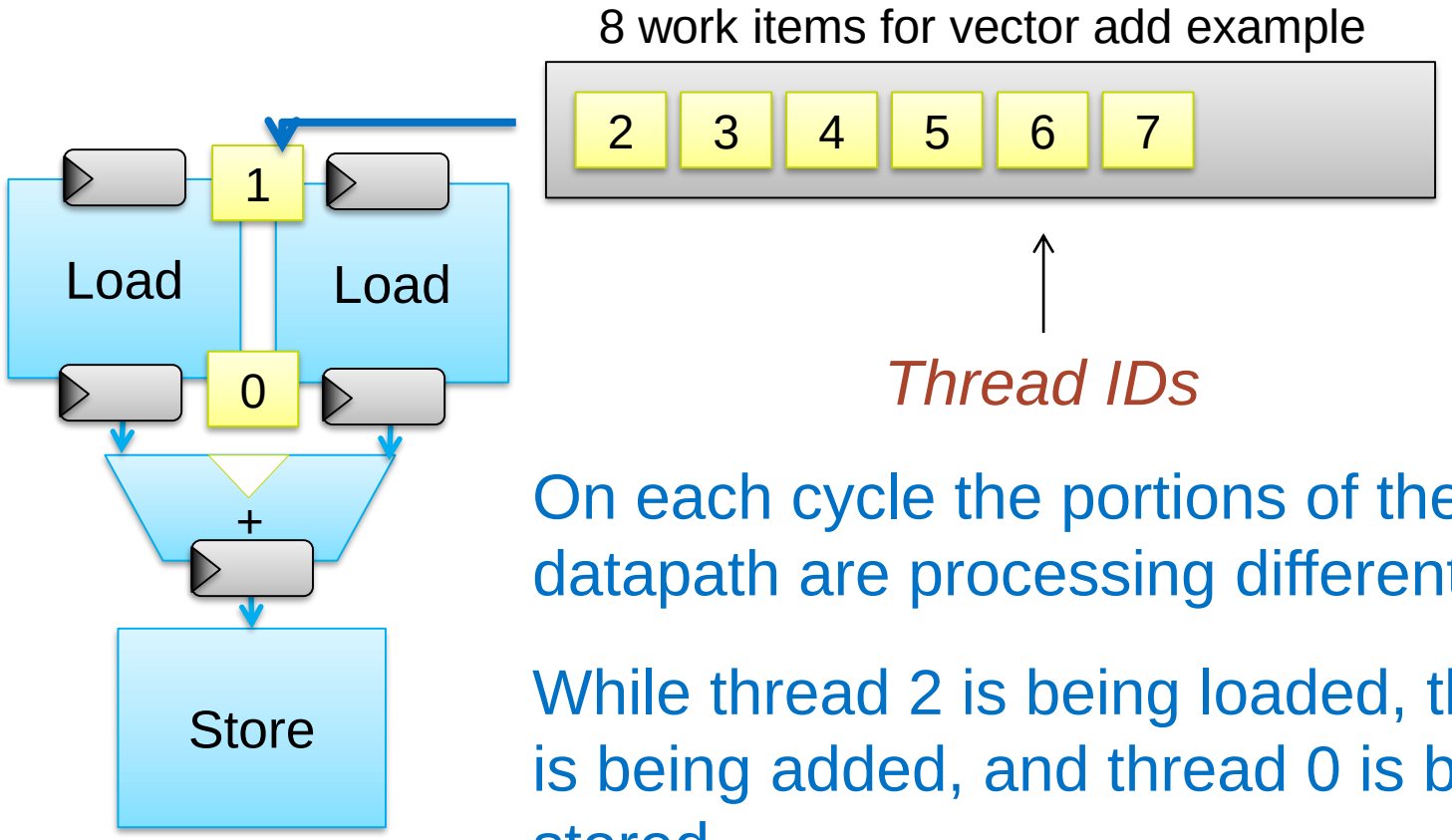
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

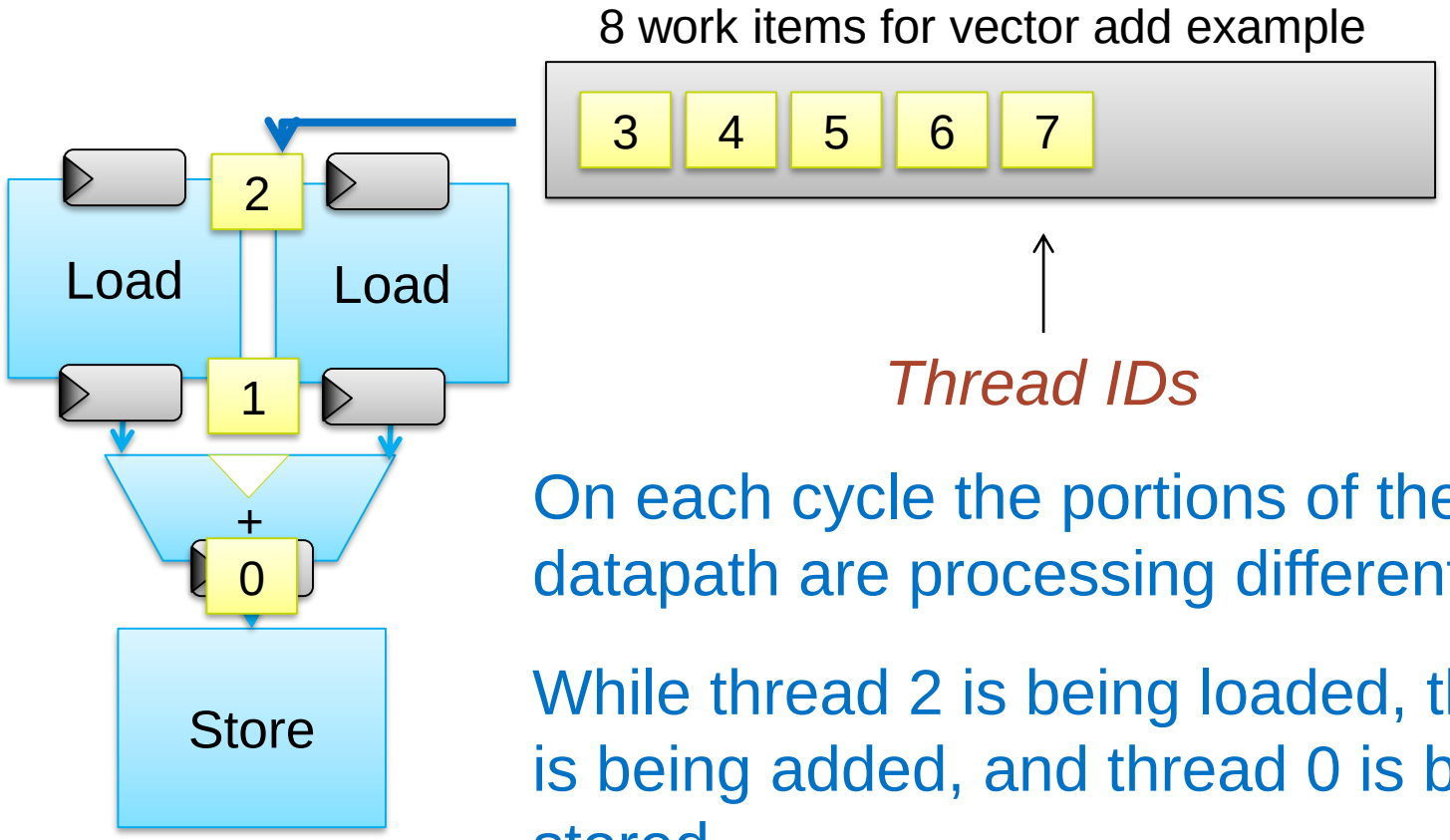
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

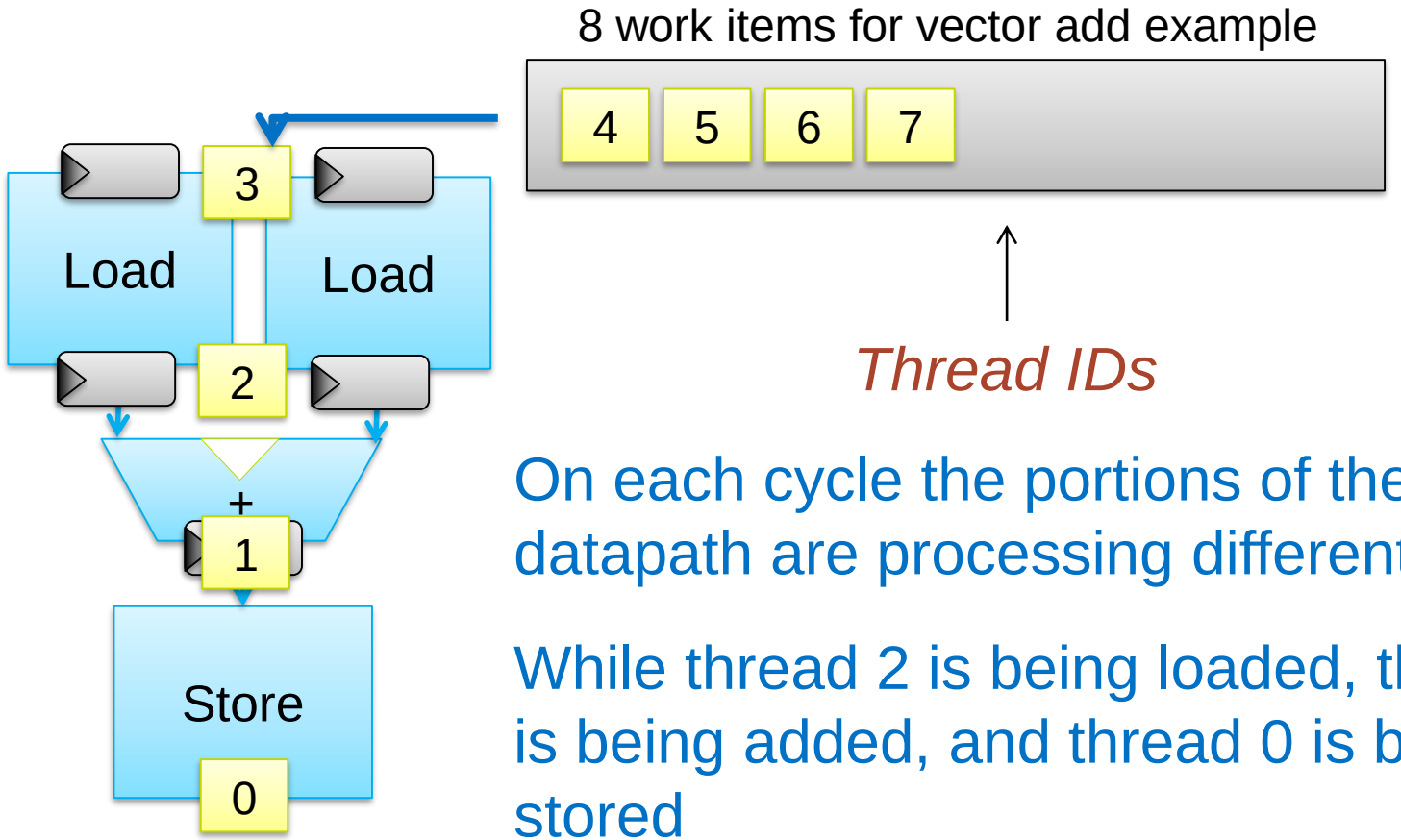
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

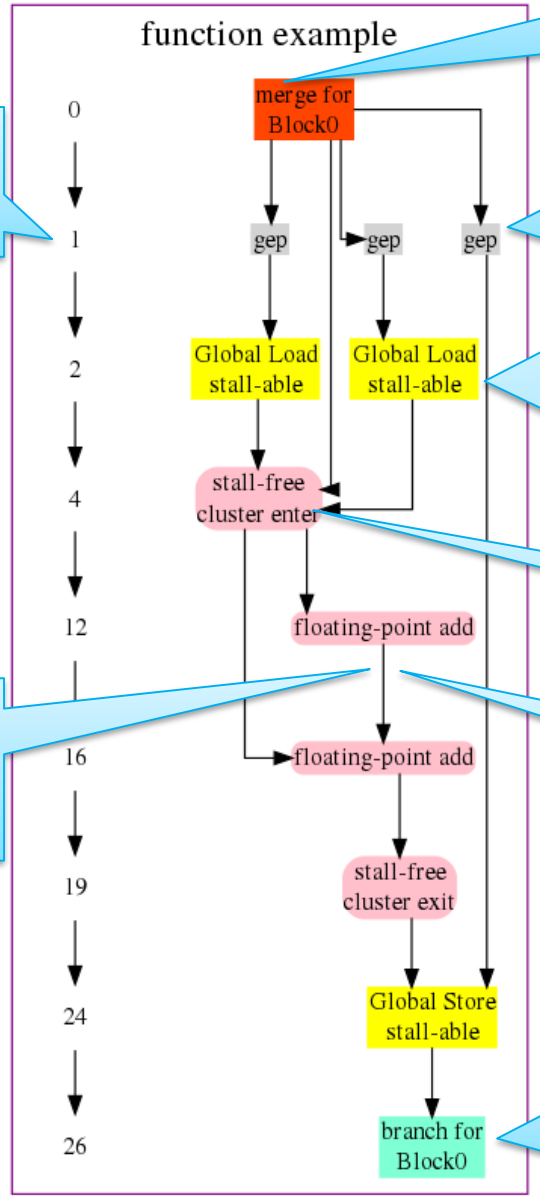
Real scheduling example

```
// OpenCL kernel
kernel void example (
     global float * restrict in,
     global float * restrict in2,
     global float * restrict out) {

  int k = get_global_id(0);
  float x = in[k];
  float y = in2[k];
  out[k] = x + (y + 1.0f);
}
```

Cycle # at which instruction starts executing.

Dependencies between instructions are constraints to the scheduler.



Hardware block containing all PHIs.

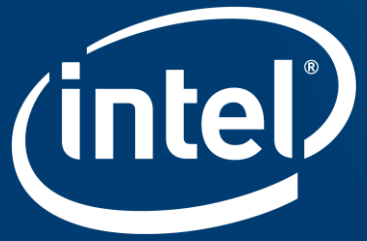
Three GEPs are executed at the same time.

Two loads (and one store below) are separate pieces of hardware, execute concurrently for this thread.

Special hardware-only optimization construct.

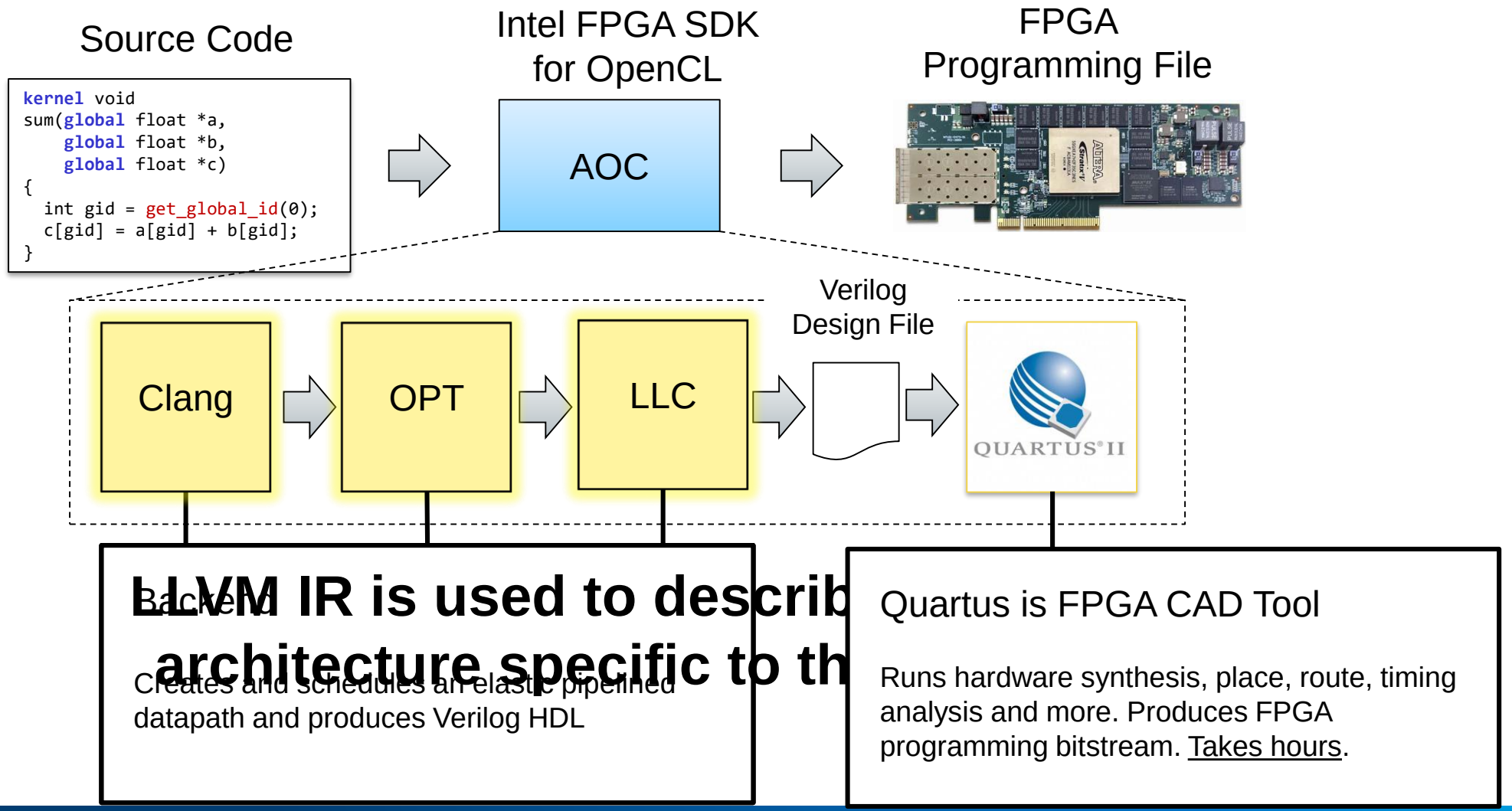
Two floating-point adders depend on each other, execute sequentially for this thread.

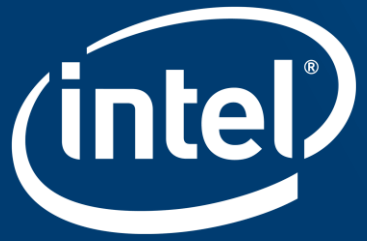
Contains inter-BB branch control or 'ret.'



Compiler Flow

Compiler Flow





Example Compiler Optimizations

Branch Conversion

Control flow is expensive.

Instead, execute both sides of a branch, pick the result for the “true” path, and predicate commands that have side-effects.

If a function has no loops, the whole function loses all branches.

Loops lose all internal branches.

```
1. X = W;
2. if (cond) {
3.     X += 2;
4.     array[z] = Y;
5. }
```

```
X_temp = X + 2;
X = cond ? X_temp : W;
array[z] = Y only if cond
```

?: operator is a mux (selector) in hardware.

Single IR instruction to store only if condition is true. “cond” is predicate on the store unit. Requires store IR instruction to accept predicate.

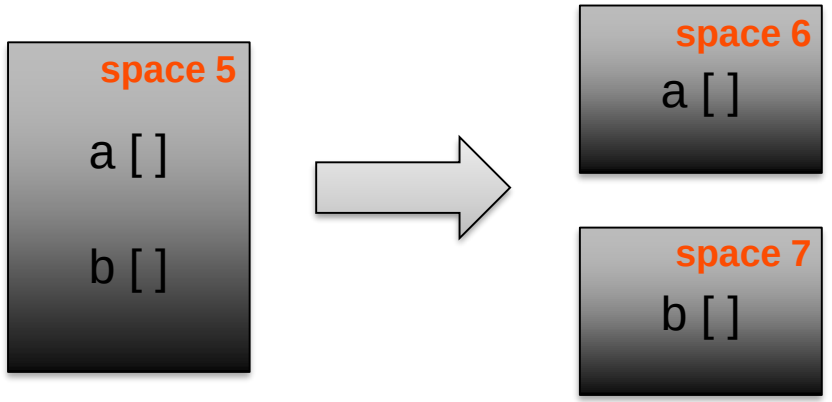
Local memory address space splitting

FPGAs lack seamless memory hierarchy that CPUs have.

We use memory address spaces to distinguish different memory locations: on-chip, off-chip, and special types of off-chip memory (e.g. constant, QDR, HMC).

On-chip (aka local memory) is further split into multiple address spaces based on access patterns for much better implementation efficiency:

```
int local_mem_user(…) {
   int a[SIZE];
   int b[SIZE];
   <code that accesses a[]>
   <code that accesses b[]>
}
```

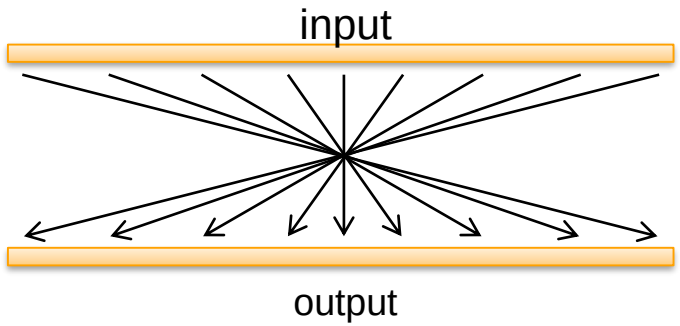


Split is possible only if compiler can prove that pointers to a[] and b[] never mix.

Optimizing Bit Swizzling

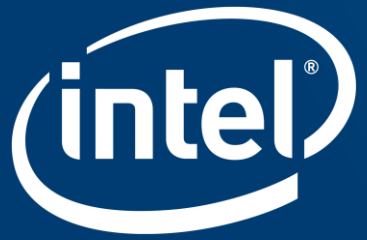
Bit swizzling with compile-time known pattern (e.g. bit reversal) is free on FPGA.

```
int bit_reverse(int x) {
    int output = 0;
    #pragma unroll
    for (int i = 0; i < 32; i++) {
        output <<= 1;
        if (x & 1) output |= 1;
        x >>= 1;
    }
    return output;
}
```



Without optimization, IR above is a very expensive tree of ORs and ANDs.

Compiler detects such an IR tree and turns it into a single `shufflevector` instruction.

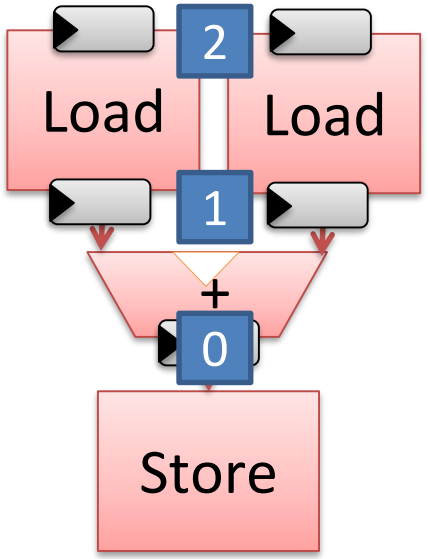


3. Loop Pipelining

Data-Parallel Execution

On the FPGA, we use pipeline parallelism to achieve acceleration

```
kernel void
sum(global const float *a,
    global const float *b,
    global float *c)
{
  int xid = get_global_id(0);
  c[xid] = a[xid] + b[xid];
}
```



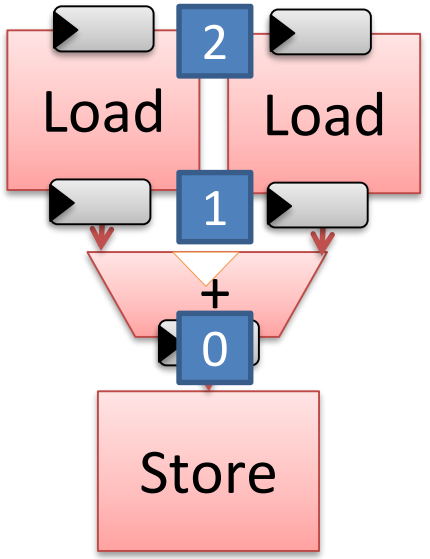
Threads execute in an embarrassingly parallel manner.

Ideally, all parts of the pipeline are active at the same time.

Data-Parallel Execution - drawbacks

Difficult to express programs which have partial dependencies during execution

```
kernel void
sum(global const float *a,
    global const float *b,
    global float *c)
{
  int xid = get_global_id(0);
  c[xid] = c[xid-1] + b[xid];
}
```



Would require complicated hardware and new language semantics to describe the desired behavior

Loop-pipelining

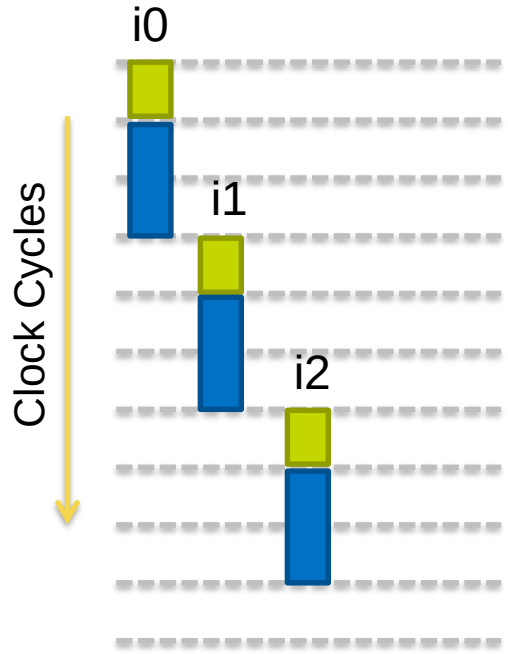
Allow users to express programs as a single-thread

```
for (int i=1; i < n; i++) {
  c[i] = c[i-1] + b[i];
}
```

Pipeline parallelism still leveraged to efficiently execute loops via **loop pipelining** – multiple loop iterations are executed concurrently.

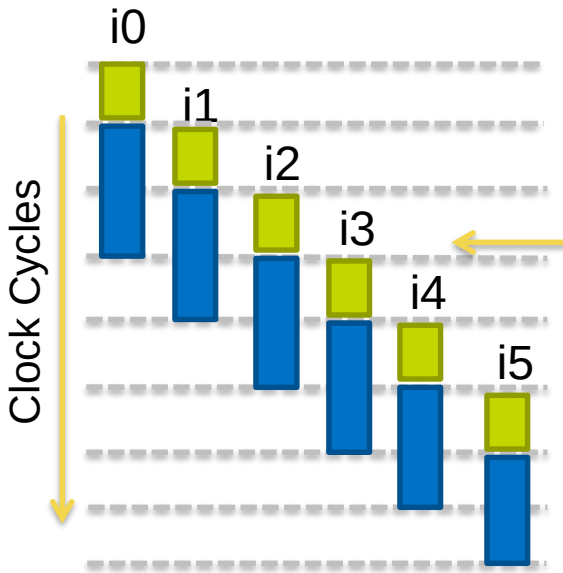
Loop Pipelining Example

No Loop Pipelining



No Overlap of Iterations!

With Loop Pipelining



Looks almost like multi-threaded execution!

Finishes Faster because Iterations Are Overlapped

Loop Pipelining enables Pipeline Parallelism AND the communication of state information between iterations.

Loop-Carried Dependencies

Loop-carried dependencies are dependencies where one iteration of the loop depends upon the results of another iteration of the loop

```
kernel void state_machine(ulong n)
{
  t_state_vector state = initial_state();
  for (ulong i=0; i<n; i++) {
    state = next_state( state );
    unit y = process( state );
    write_output(y);
  }
}
```

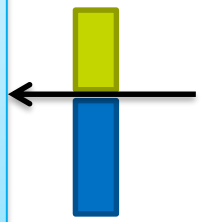
The variable state in iteration 1 depends on the value from iteration 0.

Loop-Carried Dependencies

To achieve acceleration, we pipeline each iteration of a loop with loop-carried dependencies

- Analyze any dependencies between iterations
- Schedule these operations
- Launch the next iteration as soon as the critical dependency is calculated

```
kernel void state_machine(ulong n)
{
  t_state_vector state = initial_state();
  for (ulong i=0; i<n; i++) {
    state = next_state( state );
    unit y = process( state );
    write_output(y);
  }
}
```



At this point, we can launch the next iteration

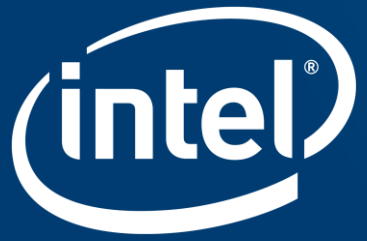
Trouble with Loop-Carried Dependencies

Many things can go wrong with loop pipelining:

- Loop-carried dependency takes too long to compute.
- Loop iterations may get out of order.

Consequences of having a loop-carried dependency are severe:

- If introduce dependency on global location: loop initialization internal can go from 1 to ~70.
  - That's 70x drop in performance!
- The compiler has to be good at analyzing and reporting these dependencies!



LLVM: Benefits & Challenges

Benefits

LLVM is awesome!

Challenges

Our instruction costs are wildly different from CPUs.

Well formed loops are extremely important to us but …

- Our ideal loop form is not the same as for CPU. Never want loops replicated or put inside a condition. Often no point in hoisting if there is no dependency.

Need many custom intrinsics to better model our hardware:

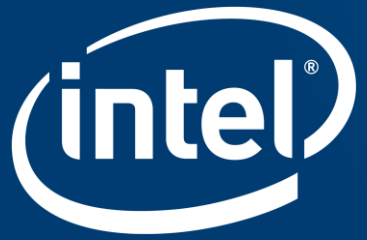
- Load/store units with additional arguments: predicates, byte-enables, dependencies.
- Channels to express communication between parallel tasks.

Have to use debug data to carry additional information:

- Have “styles” of load/store units and even multipliers (high throughput, low area)

Can’t express instruction-level, block-level, and task-level parallelisms:

- Only decide on this in the backend, and it’s late or very expensive to do optimizations then.



We're hiring!

Thank You