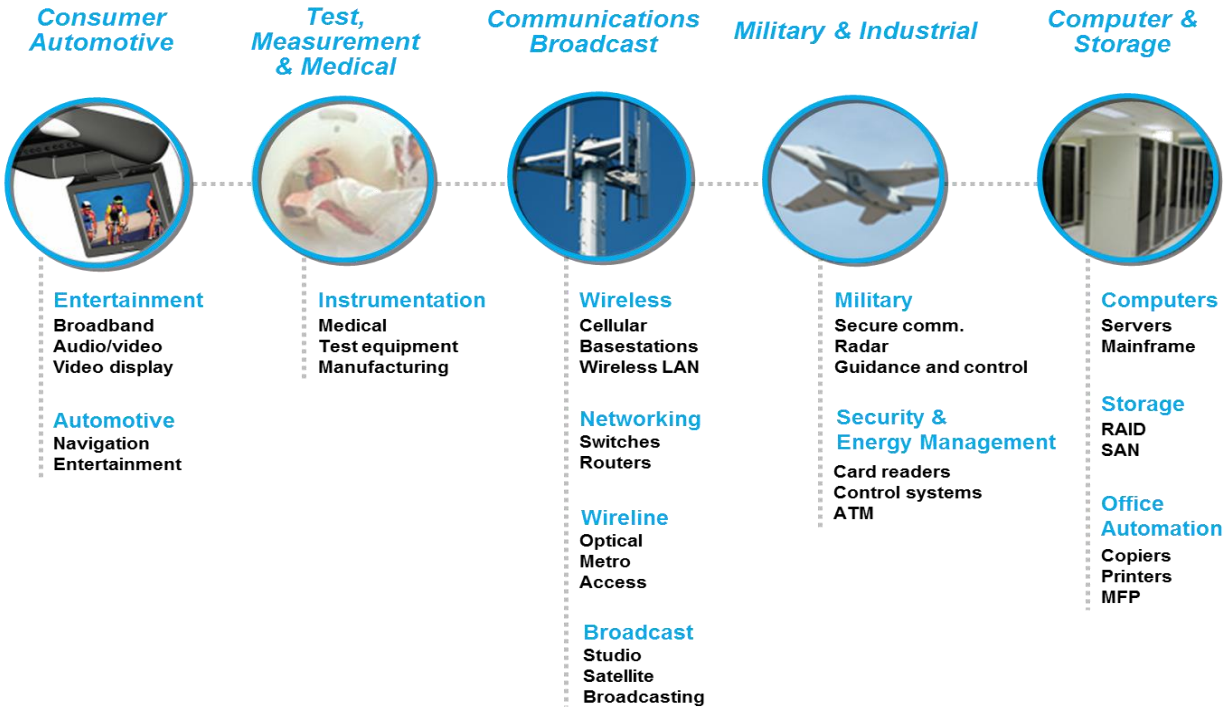
Custom Hardware State-Machines and Datapaths – Using LLVM to Generate FPGA Accelerators

Alan Baker

Altera Corporation

FPGAs are Awesome

- **Fully Configurable Architecture**
- **Low-Power**
- **Customizable I/O**



ALTERA®
MEASURABLE ADVANTAGE™

FPGA Design Hurdles

- **Traditional FPGA design entry done in hardware description languages (HDL)**
  - e.g. Verilog or VHDL
  - HDL describe the register transfer level (RTL)
  - Programmer is responsible for describing all the hardware and its behaviour in every clock cycle
  - The hardware to describe a relatively small program can take months to implement
  - Testing is difficult
- **Far fewer hardware designers than software designers**

ALTERA
MEASURABLE ADVANTAGE™

Simpler Design Entry

- **Use a higher level of abstraction**
  - Easier to describe an algorithm in C than Verilog
  - Increases productivity
  - Simpler to test and verify
  - Increases the size of the developer pool

- **Sounds promising, but how can we map a higher level language to an FPGA?**

ALTERA
MEASURABLE ADVANTAGE™

Our Vision

- **Leverage the software community's resources**

- **LLVM is a great compiler framework**
  - Mature
  - Robust
  - Well architected
  - Easy to modify and extend
  - Same IR for different input languages

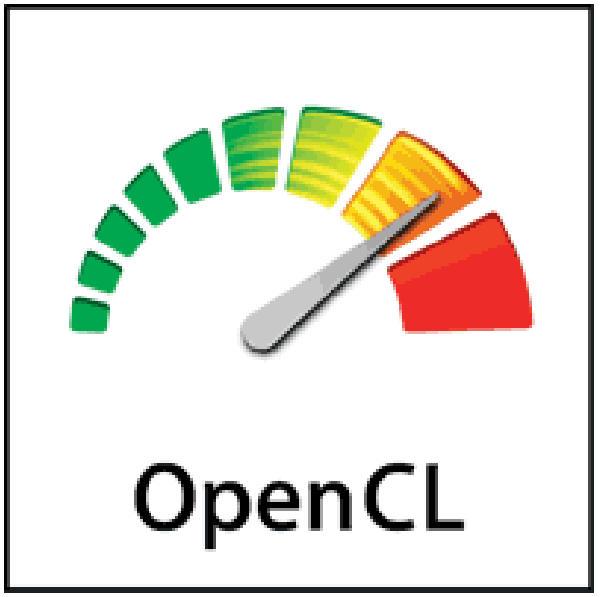
- **We modify LLVM to generate Verilog**
  - Implemented a custom backend target

ALTERA
MEASURABLE ADVANTAGE™

OpenCL

- **Our higher level language**
- **Hardware agnostic compute language**
  - Invented by Apple
  - 2008 Specification Donated to Khronos Group and Khronos Compute Working Group was formed

KHRONOS GROUP



- **What does OpenCL give us?**
  - Industry standard programming model
  - Aimed at heterogeneous compute acceleration
  - Functional portability across platforms

ALTERA
MEASURABLE ADVANTAGE™

OpenCL Conformance

- **You must pass conformance to claim OpenCL support**
  - Over **8000** tests
  - *Only one FPGA vendor has passed conformance*

ALTERA
MEASURABLE ADVANTAGE™

The BIG Idea behind OpenCL

- **OpenCL execution model …**
  - Define N-dimensional computation domain
  - Execute a kernel at each point in computation domain

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

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**

ALTERA
MEASURABLE ADVANTAGE™

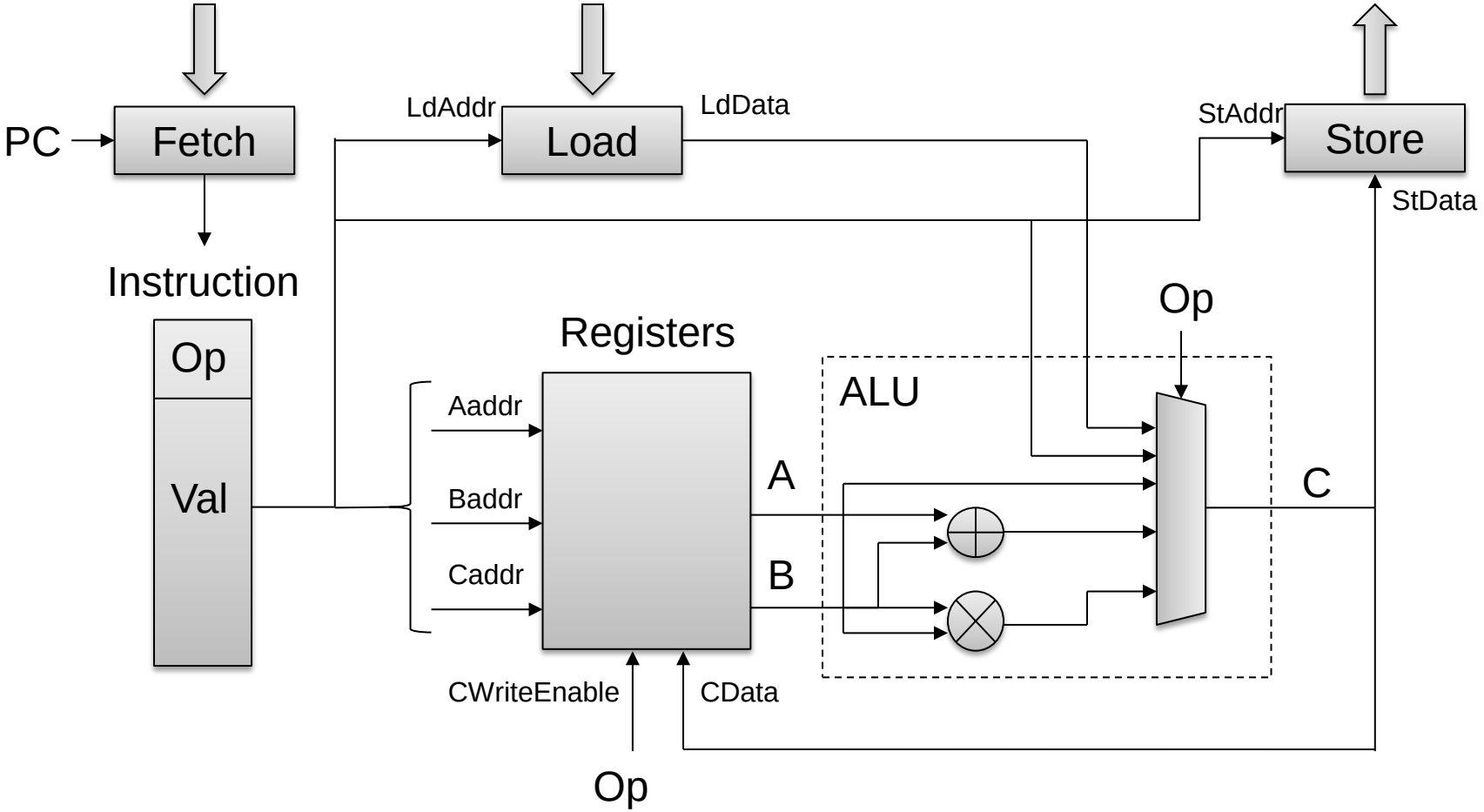
Targeting an Architecture

- In a CPU, the program is mapped to a fixed architecture
- In an FPGA, there is NO fixed architecture
- The program defines the architecture
- Instead of the architecture constraining the program, the program is constrained by the available resources

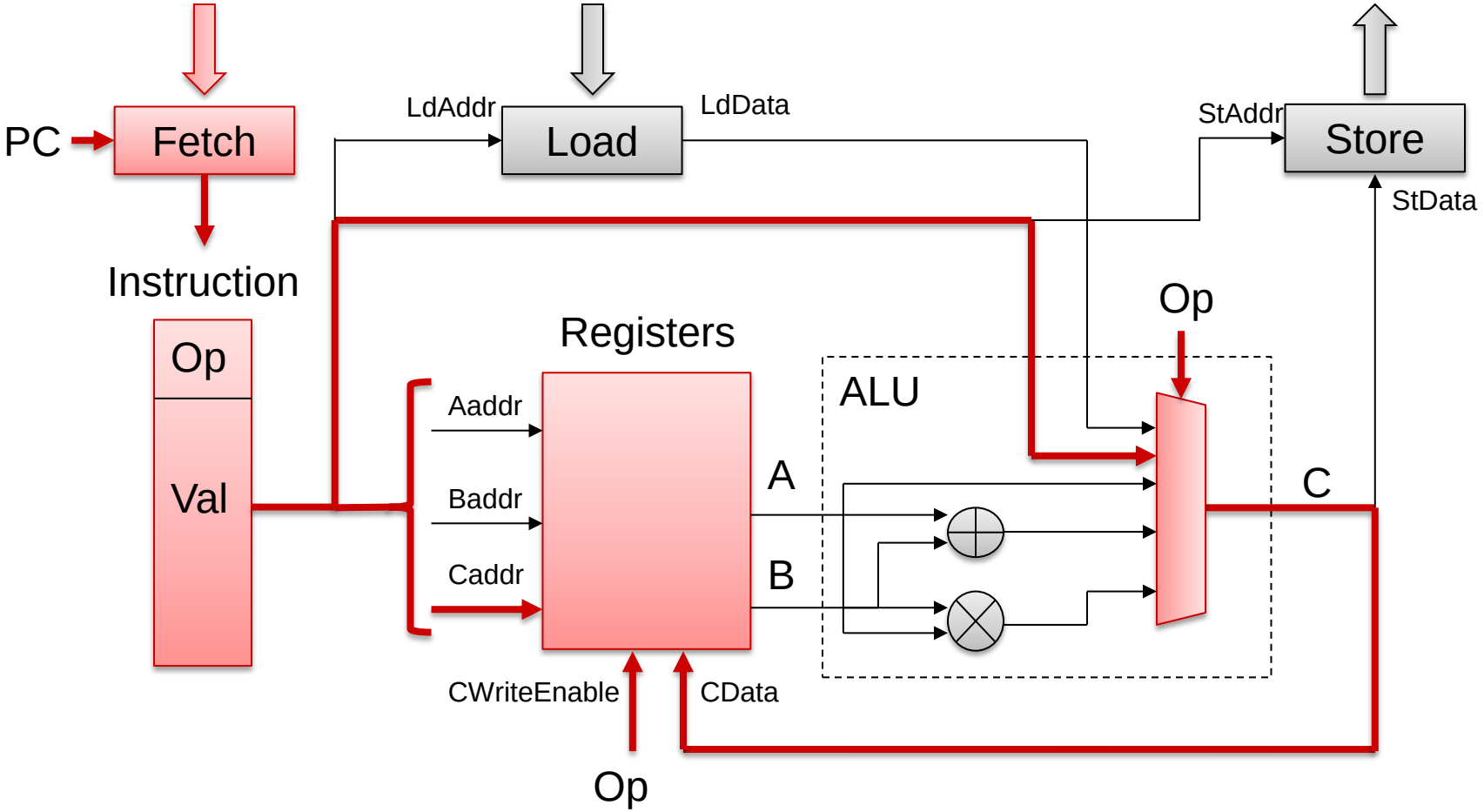
Datapath Architecture

FPGA datapath ~ Unrolled CPU hardware

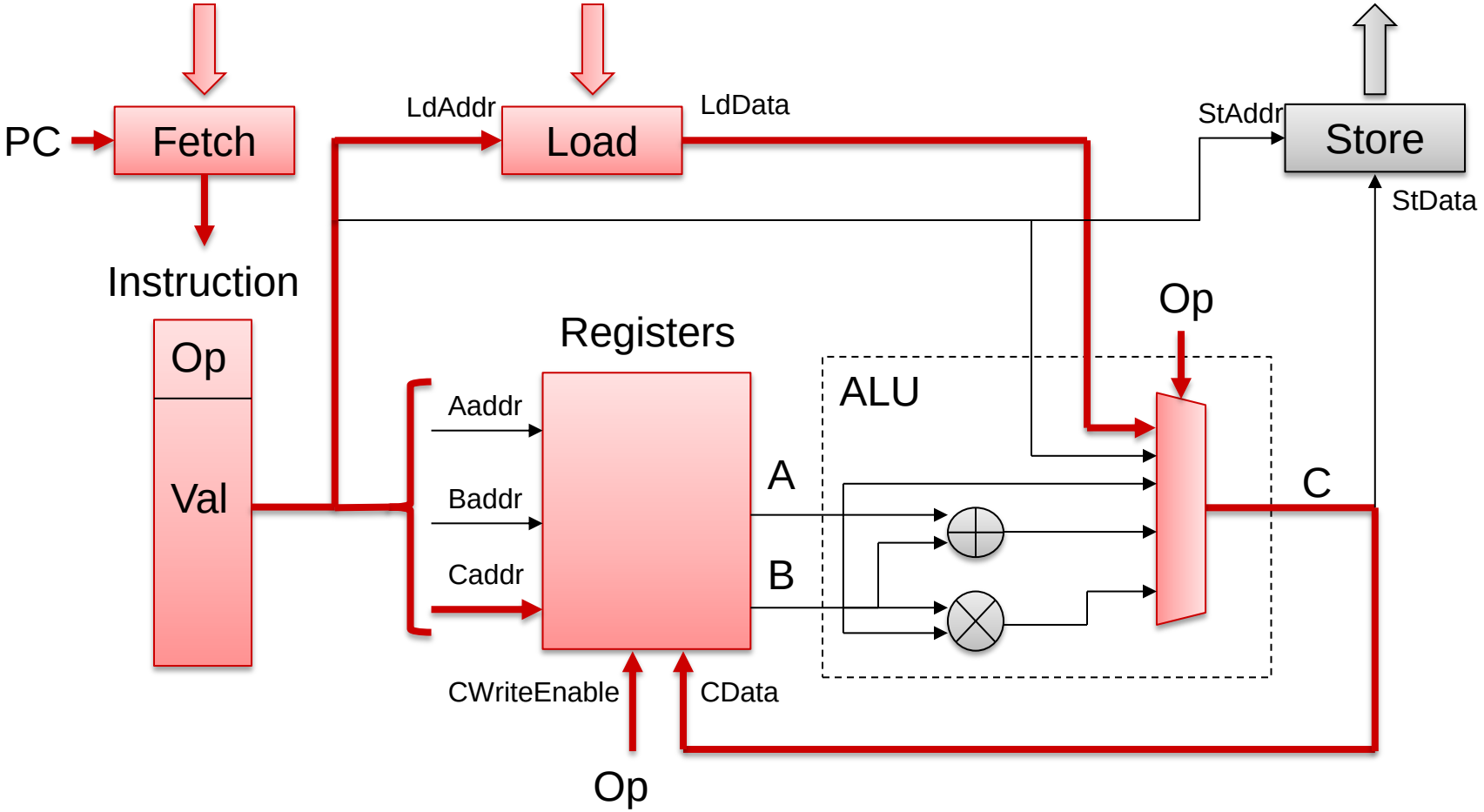
A simple 3-address CPU



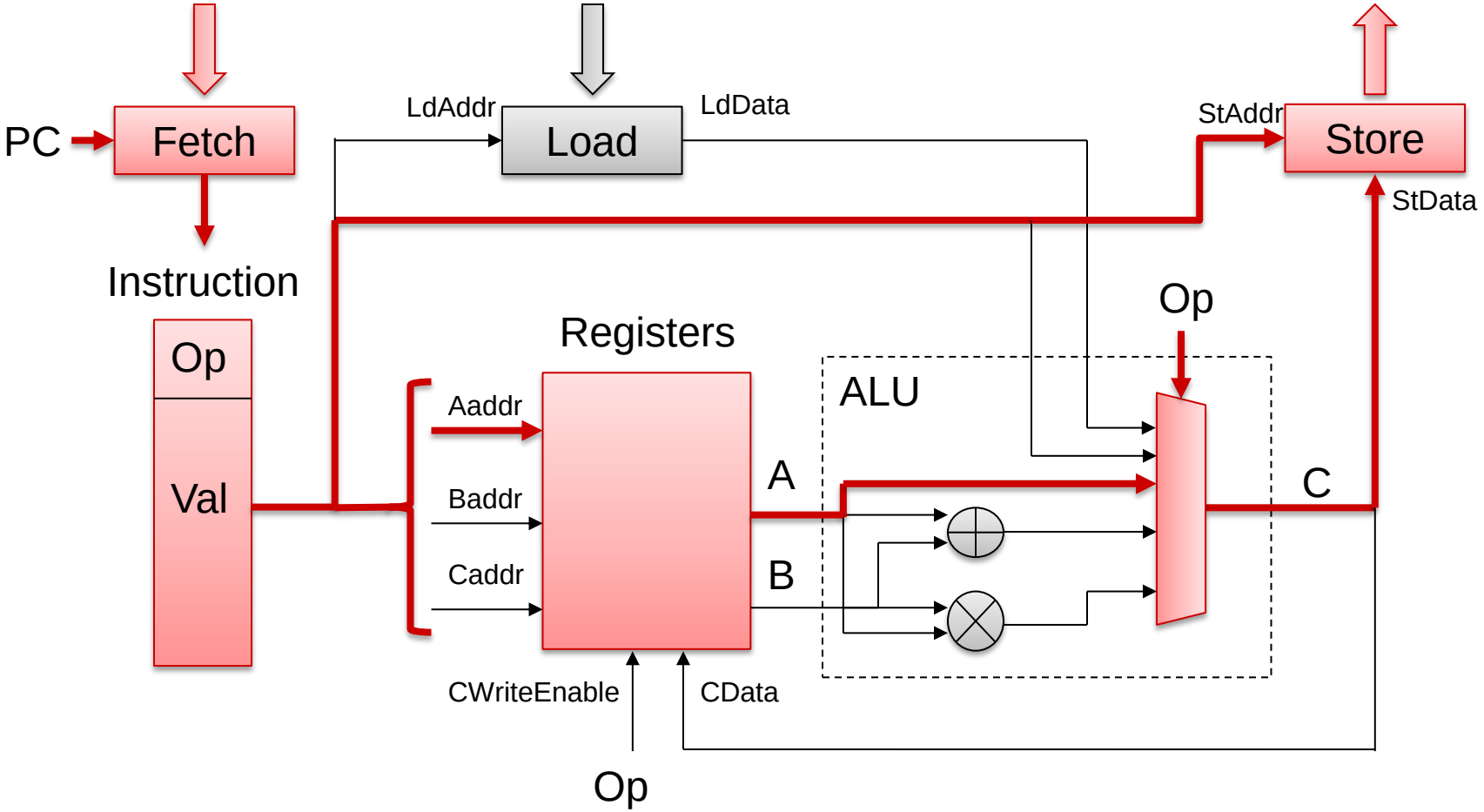
Load immediate value into register



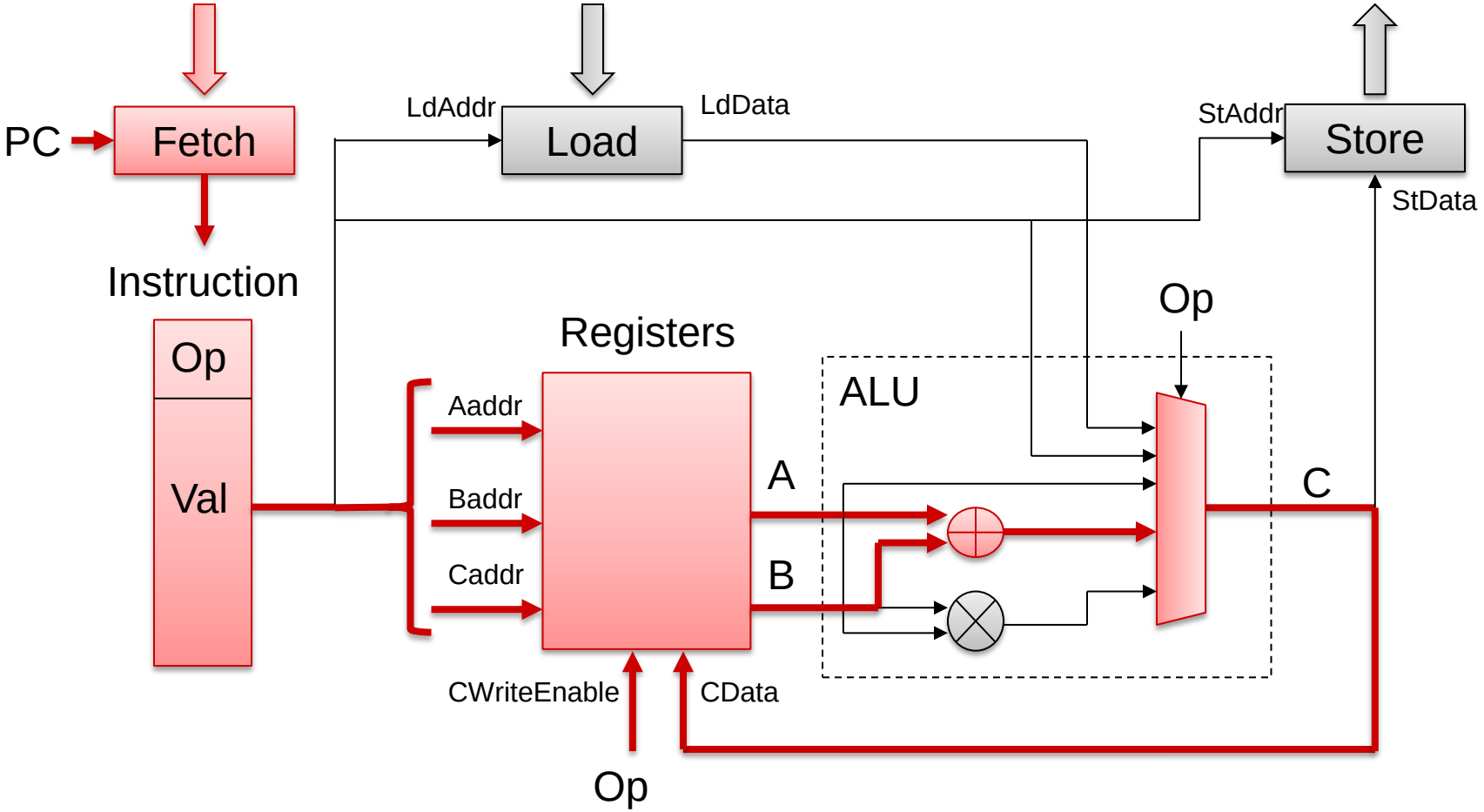
Load memory value into register



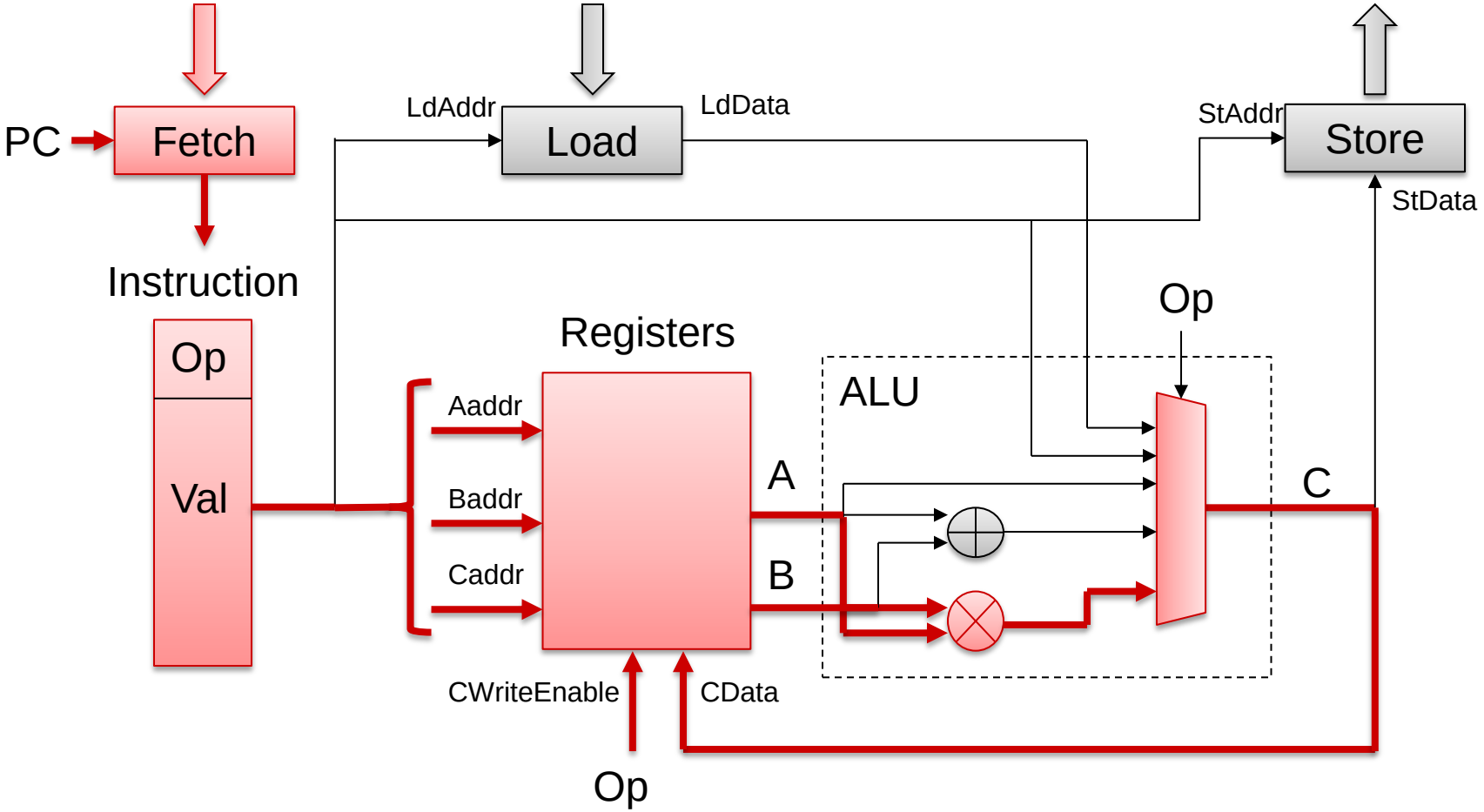
Store register value into memory



Add two registers, store result in register



Multiply 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]

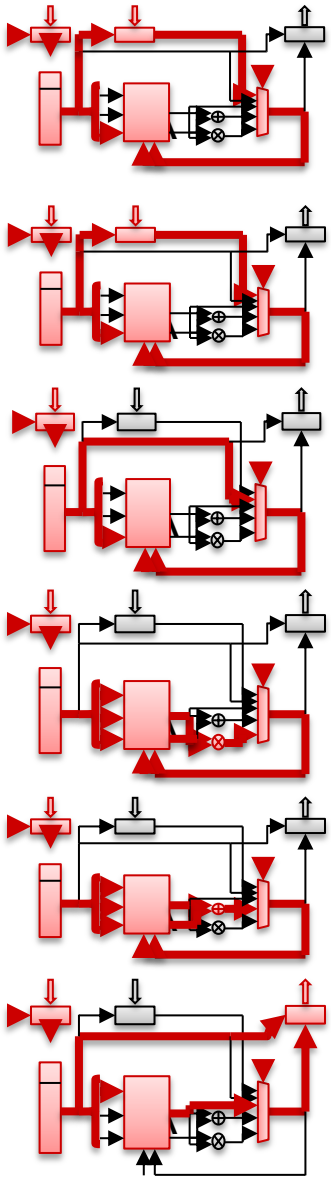
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]



Time

ALTERA
MEASURABLE ADVANTAGE™

Unroll the CPU hardware…

R0 ← Load Mem[100]

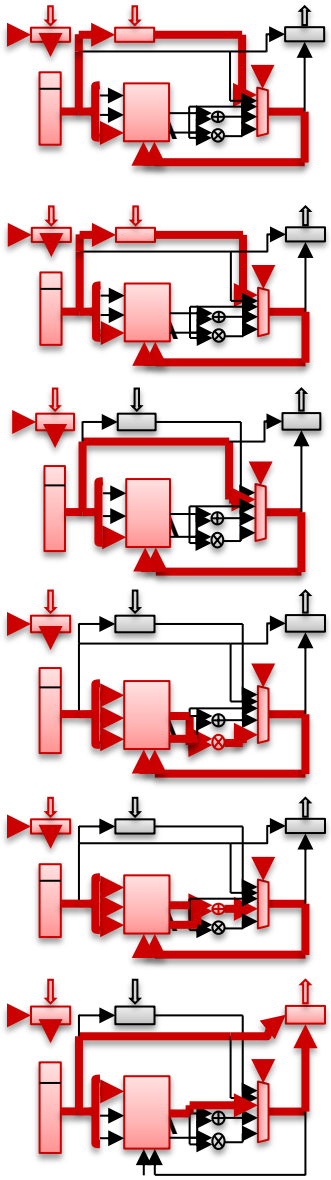
R1 ← Load Mem[101]

R2 ← Load #42

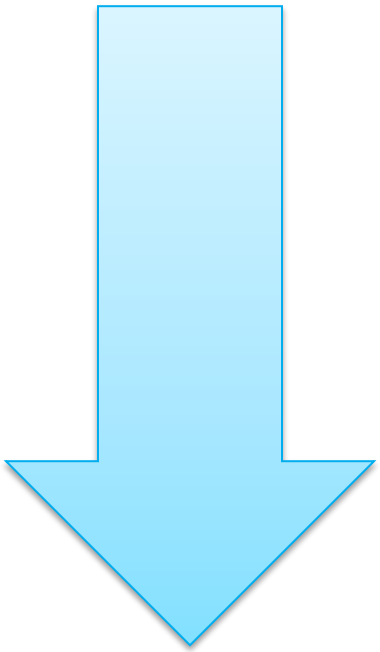
R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]



Space



ALTERA
MEASURABLE ADVANTAGE™

… and specialize by position

R0 ← Load Mem[100]

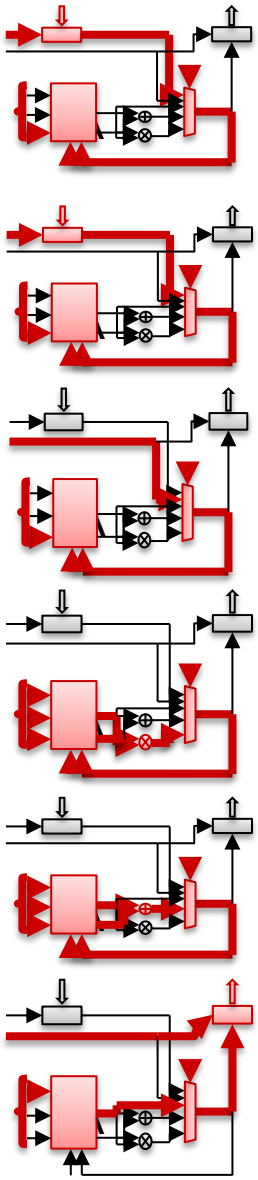
R1 ← Load Mem[101]

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]

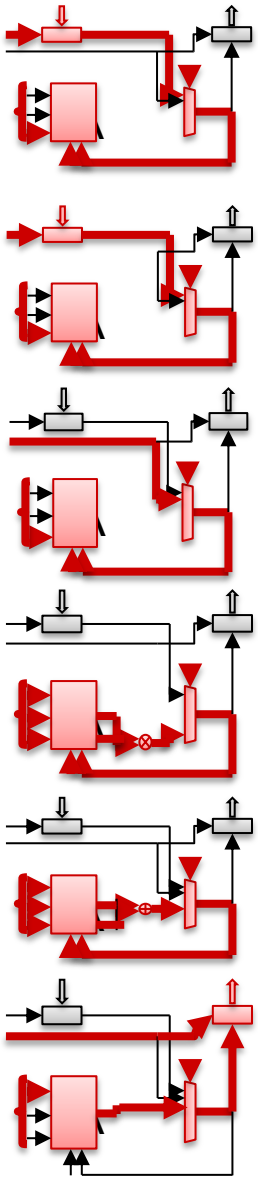
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]

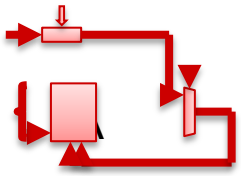


1. Instructions are fixed. Remove “Fetch”
2. Remove unused ALU ops

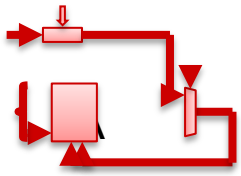
ALTERA
MEASURABLE ADVANTAGE™

… and specialize

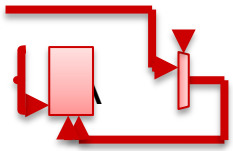
R0 ← Load Mem[100]



R1 ← Load Mem[101]

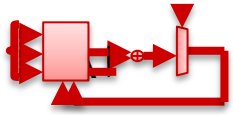


R2 ← Load #42

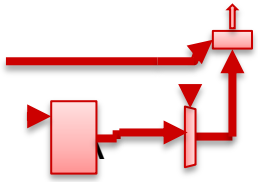


R2 ← Mul R1, R2

R0 ← Add R2, R0



Store R0 → Mem[100]



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

ALTERA
MEASURABLE ADVANTAGE™

… and specialize

R0 ← Load Mem[100]

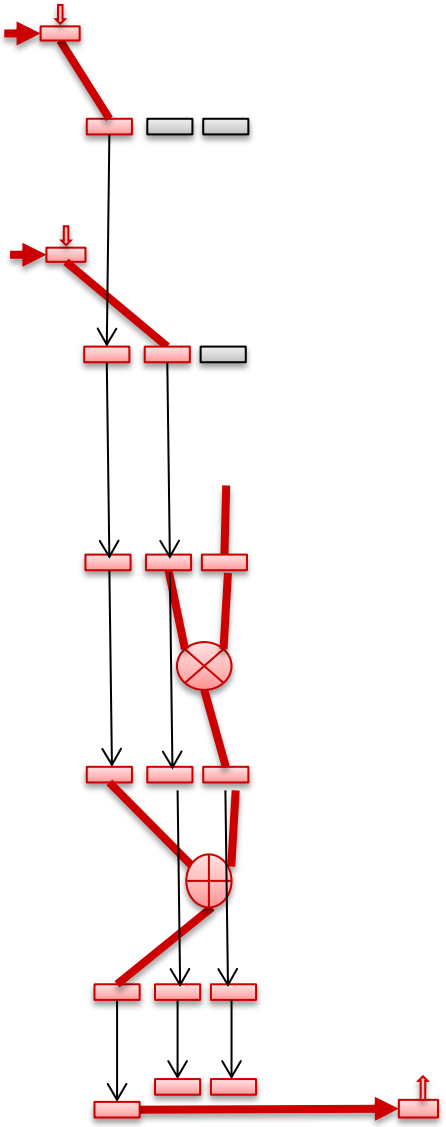
R1 ← Load Mem[101]

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.

ALTERA
MEASURABLE ADVANTAGE™

… and specialize

R0 ← Load Mem[100]

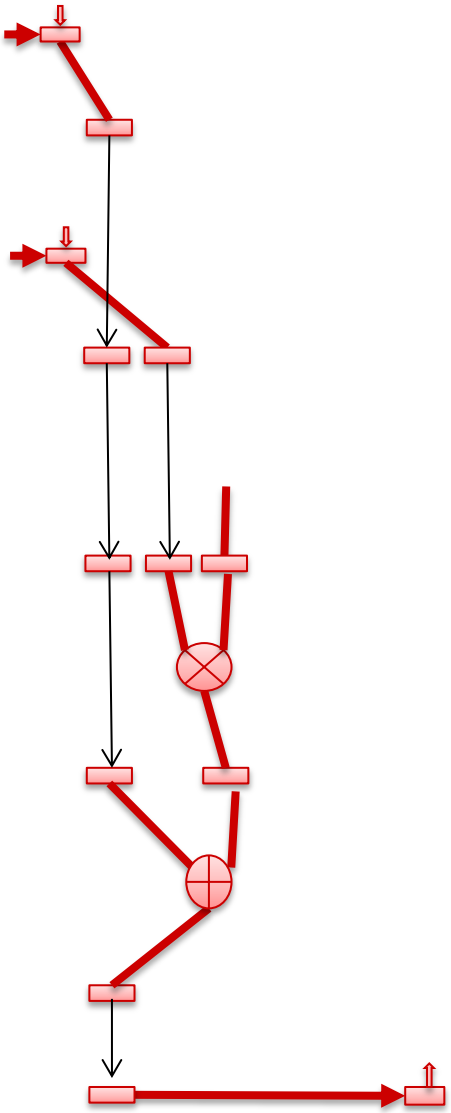
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

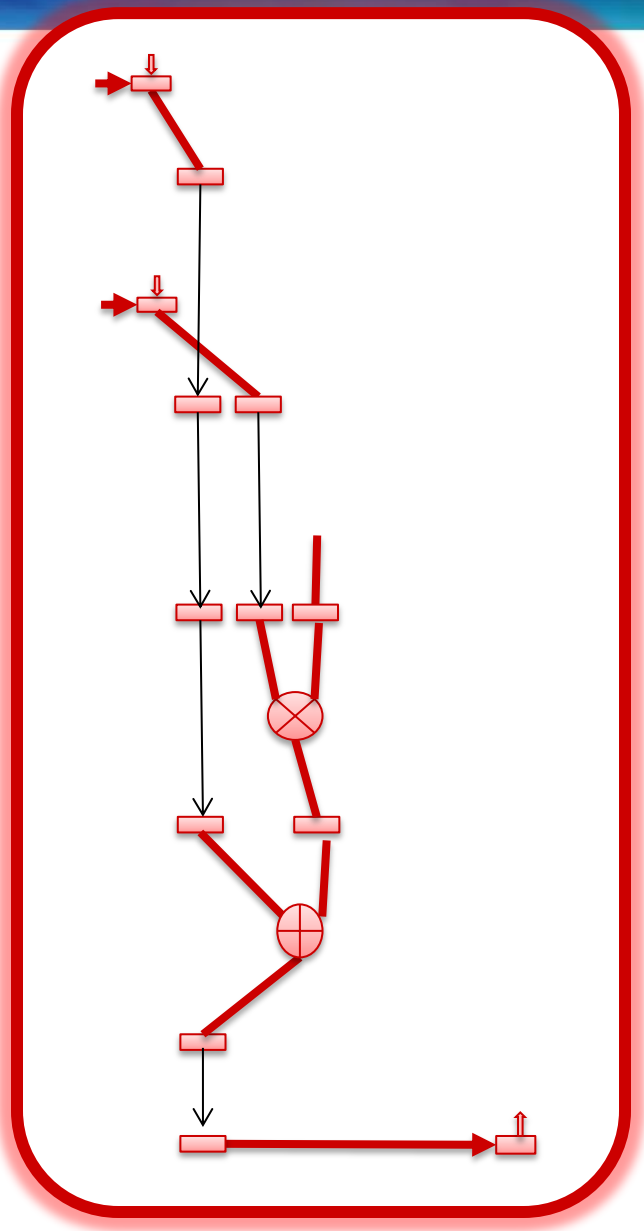
Store R0 → Mem[100]



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.

ALTERA
MEASURABLE ADVANTAGE™

Fundamental Datapath



Instead of a register file, live data is carried through register stages like a pipelined CPU instruction

Live ranges define the amount of data carried at each register stage

Optimize the Datapath

R0 ← Load Mem[100]

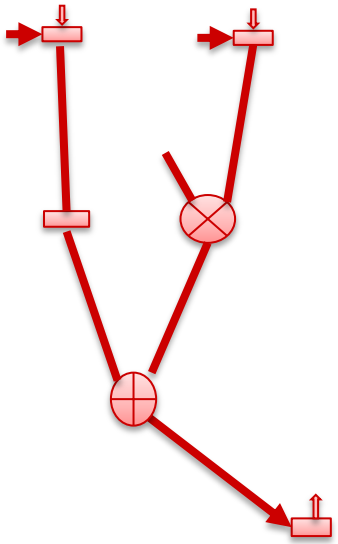
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

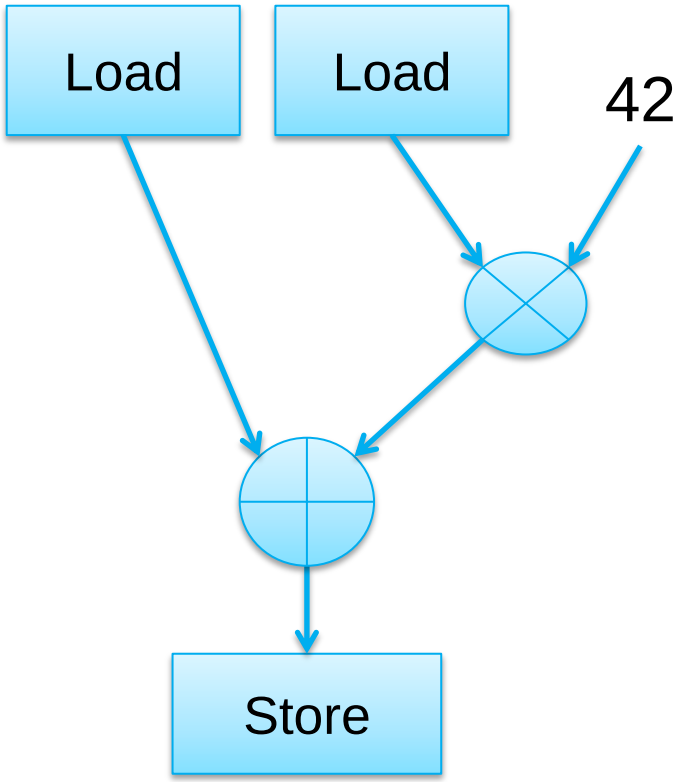
Store R0 → Mem[100]



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!

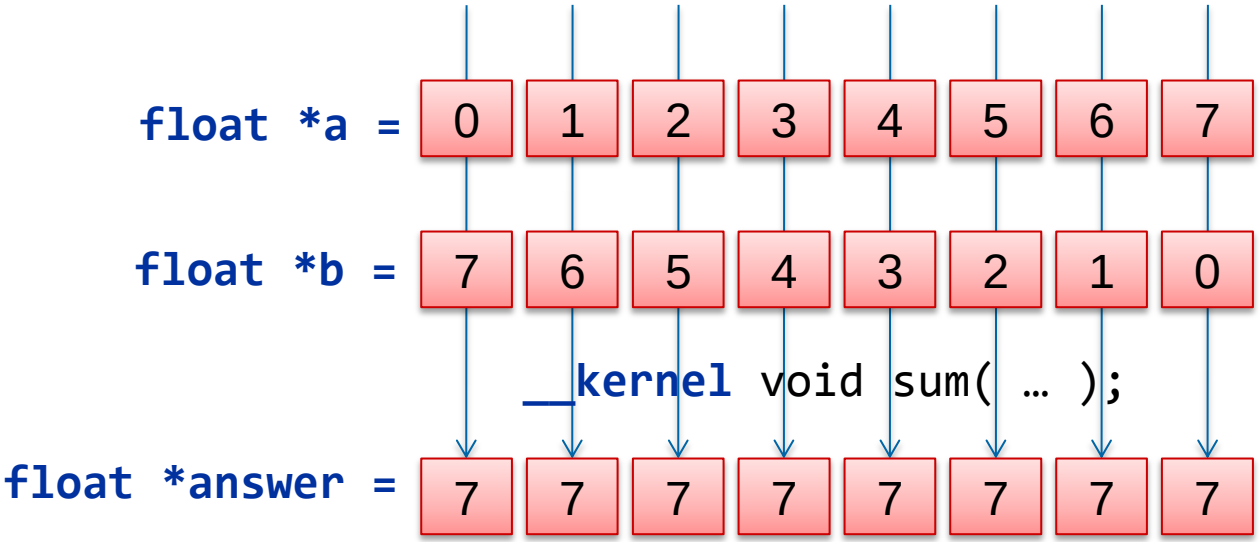
ALTERA
MEASURABLE ADVANTAGE™

FPGA datapath = Your algorithm, in silicon



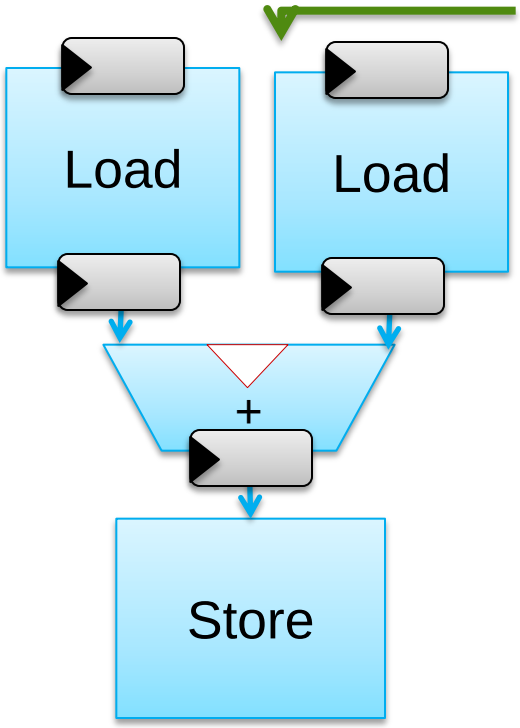
Data parallel kernel

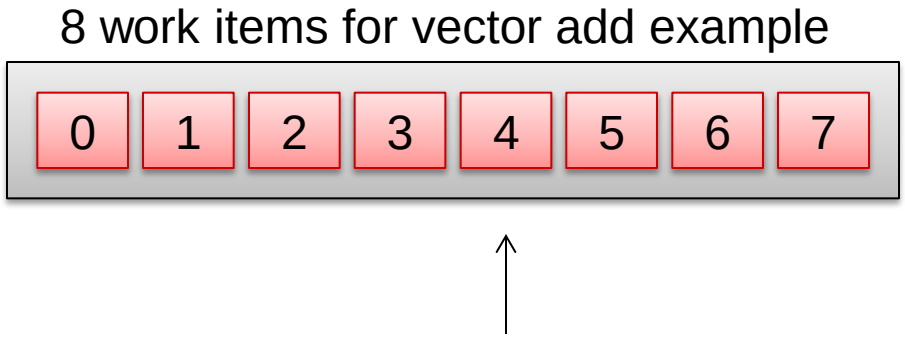
```
__kernel void
sum(__global const float *a,
__global const float *b,
__global float *answer)
{
int xid = get_global_id(0);
answer[xid] = a[xid] + b[xid];
}
```



ALTERA
MEASURABLE ADVANTAGE™

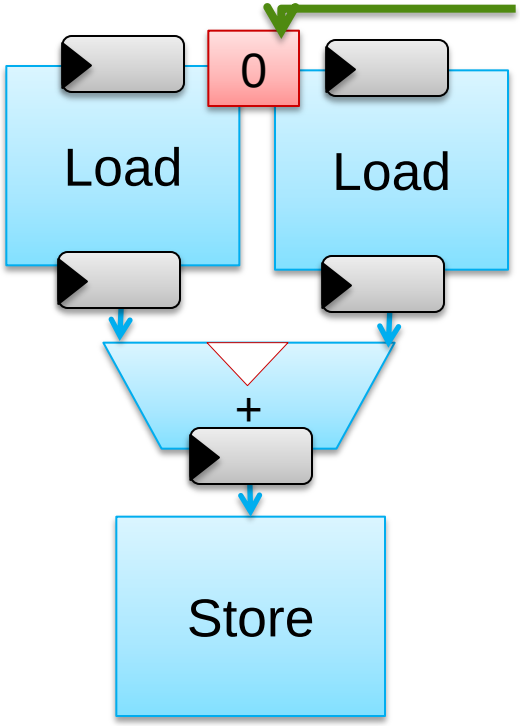
Example Datapath for Vector Add

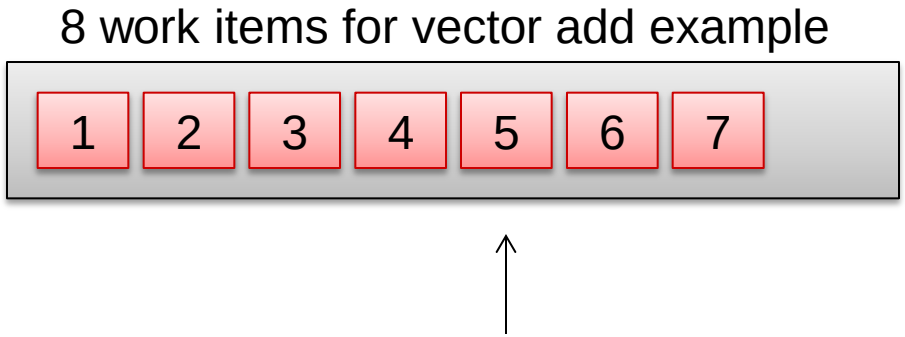




- **On each cycle the portions of the datapath are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

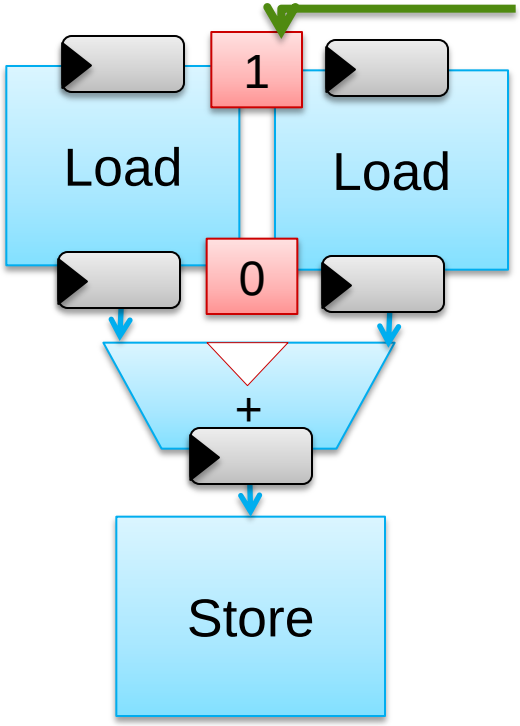
Example Datapath for Vector Add

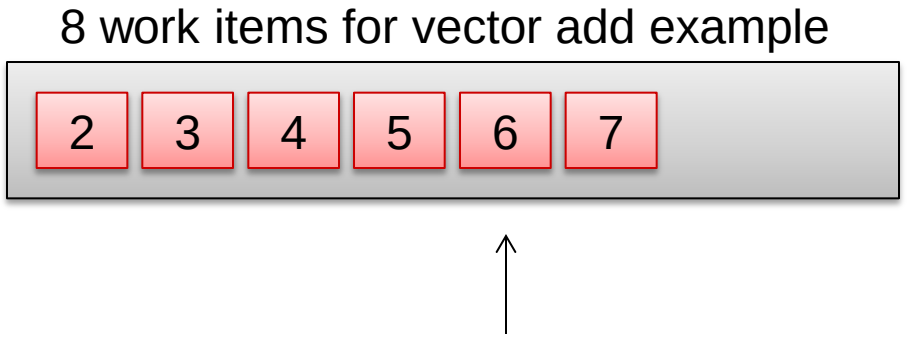




- **On each cycle the portions of the datapath are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

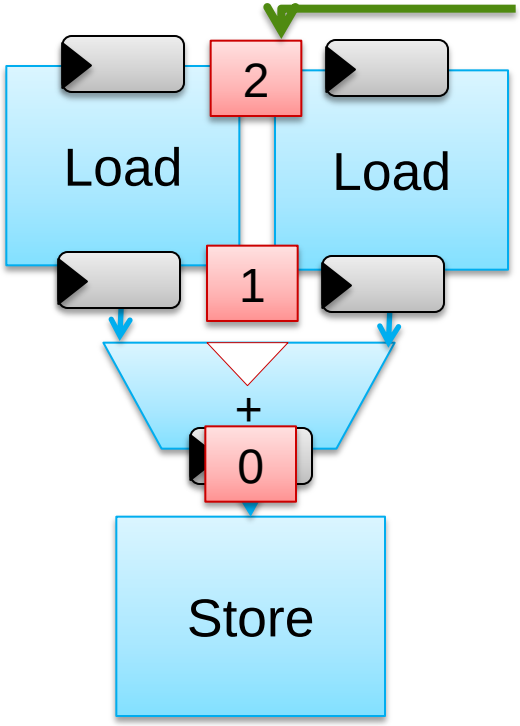
Example Datapath for Vector Add

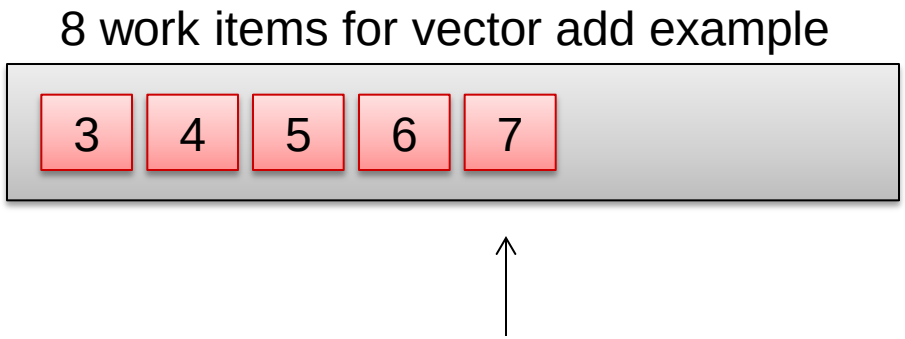




- **On each cycle the portions of the datapath are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

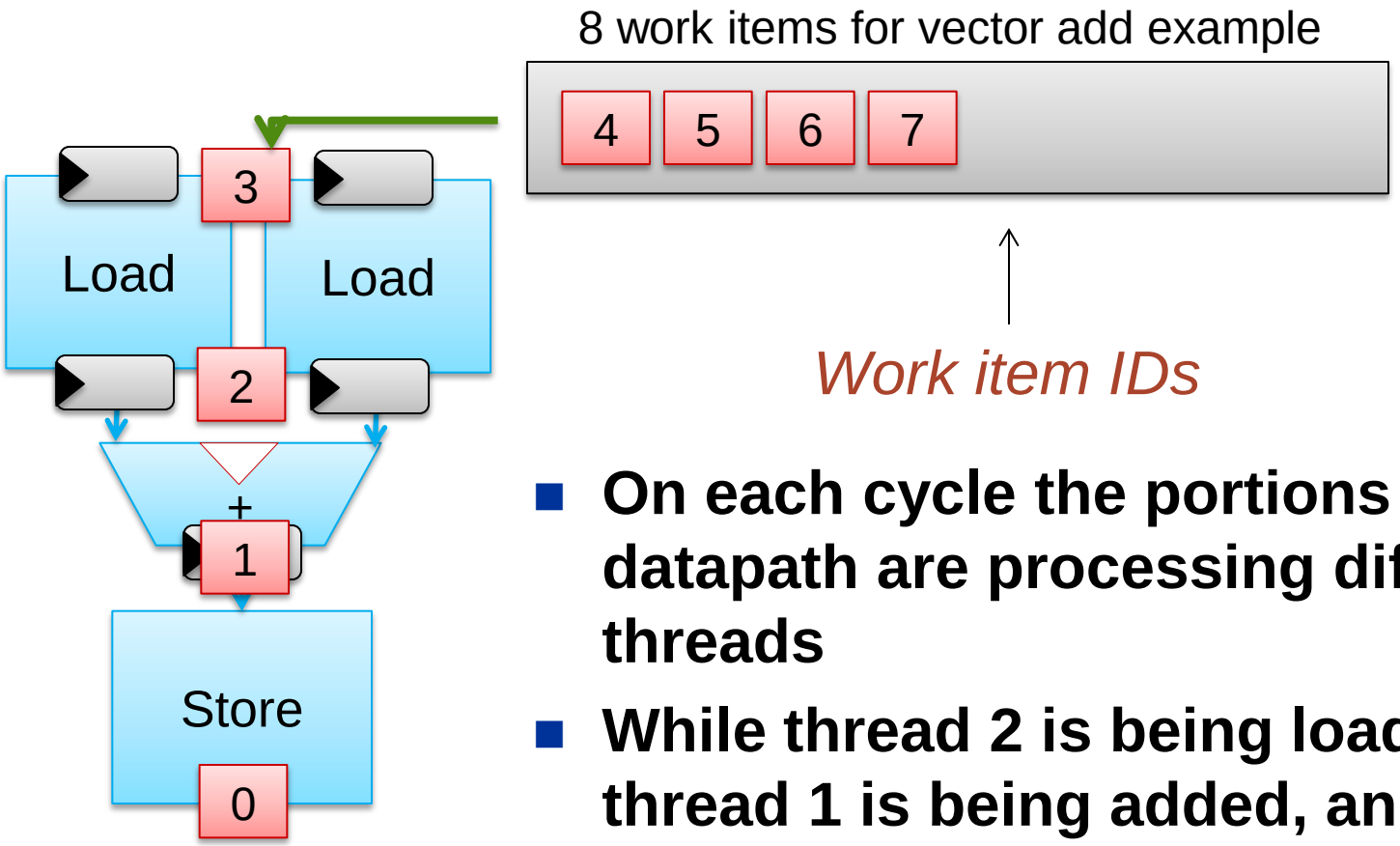
Example Datapath for Vector Add





- **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**

Silicon used efficiently at steady-state

High Level Datapath Generation

Compiler Flow

Compiler Flow

Source Code

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

Altera Offline Compiler

AOC

FPGA Programming File

Clang → OPT → LLC → Verilog Design File → QUARTUS II

Compiler Flow

Source Code

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

Altera Offline Compiler

AOC

FPGA Programming File

Clang

OPT

LLC

Verilog Design File

QUARTUS II

Frontend

Parses OpenCL extensions and intrinsics to produce LLVM IR

ALTERA MEASURABLE ADVANTAGE™

Compiler Flow

Source Code

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

Altera Offline Compiler

AOC

FPGA Programming File

Clang → OPT → LLC → Verilog Design File → QUARTUS II

Middle end

Clang –O3 optimizations followed by numerous custom passes to target the FPGA architecture

ALTERA MEASURABLE ADVANTAGE™

Compiler Flow

Source Code

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

Altera Offline Compiler

AOC

FPGA Programming File

Clang

OPT

LLC

Verilog Design File

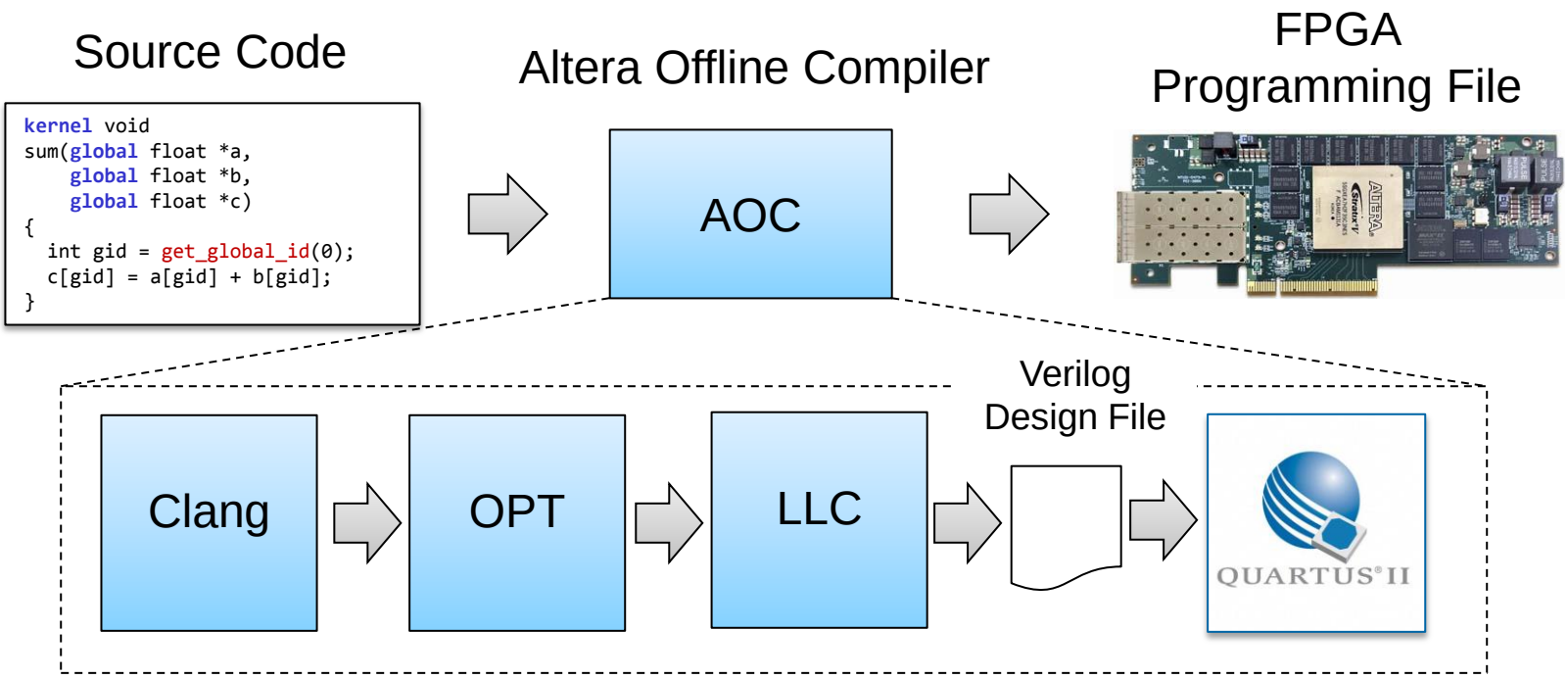
QUARTUS II

Backend

Creates and schedules an elastic pipelined datapath and produces Verilog HDL

ALTERA MEASURABLE ADVANTAGE™

Compiler Flow

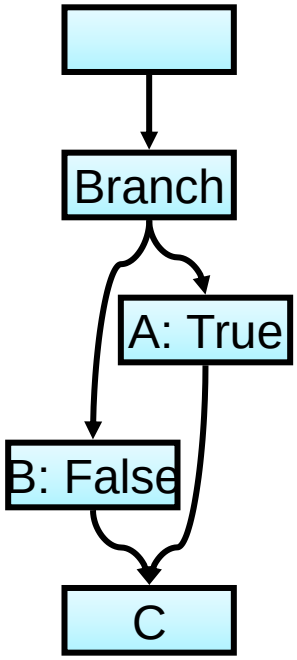


LLVM IR is used to describe a custom architecture specific to the program

Dealing with Resource Constraints

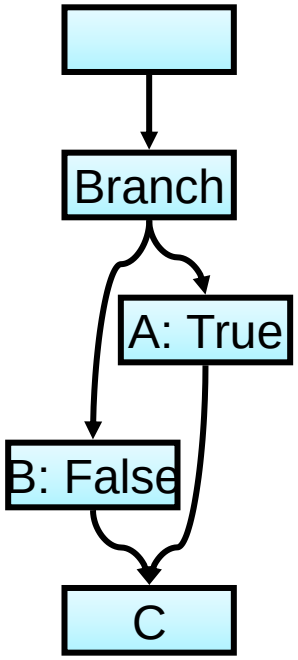
Branch Conversion

Branch Conversion Example



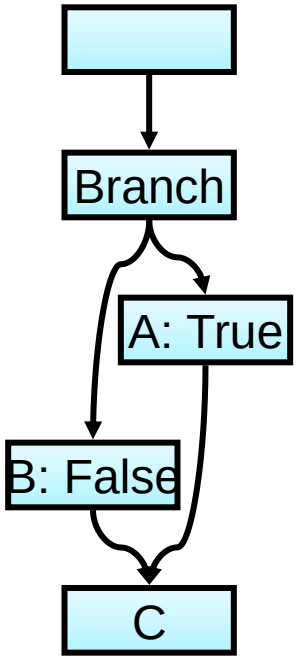
Branch Conversion Example

1. **Determine control flow to conditionally executed basic blocks**



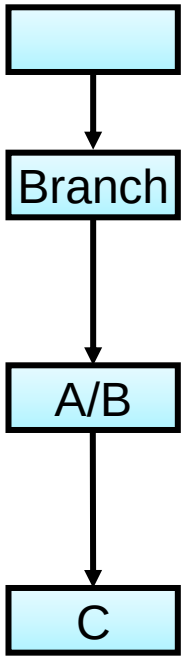
Branch Conversion Example

1. **Determine control flow to conditionally executed basic blocks**
2. **Predicate instructions**
   - A is predicated if the branch was false and vice-versa



Branch Conversion Example

1. **Determine control flow to conditionally executed basic blocks**
2. **Predicate instructions**
   - A is predicated if the branch was false and vice-versa
3. **Combine A and B**
   - Branch is now unconditional
   - PHIs in C become select instructions



Branch Conversion Example

1. **Determine control flow to conditionally executed basic blocks**
2. **Predicate instructions**
   - A is predicated if the branch was false and vice-versa
3. **Combine A and B**
   - Branch is now unconditional
   - PHIs in C become select instructions
4. **Simplify the CFG**
   - Merges remaining blocks

All Logic

Branch Conversion

- Squeezes the majority of the CFG into one basic block
- Saves significant amounts of area
- Increased instruction count in the basic block does not adversely affect performance

Improving Performance of Individual Threads

Loop Pipelining

OpenCL Task

```
__kernel void
accumulate(__global float *a,
           __global float *b,
           int n)
{
  for (int i=1; i<n; ++i)
    b[i] = b[i-1] + a[i];
}
```

- Kernel operates on a single thread
- Data for each iteration depends on the previous iteration
- Loop carried dependency bottlenecks performance

ALTERA
MEASURABLE ADVANTAGE™

Loop Carried Dependencies

- **Loop-carried dependency: 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 );
    // more work…
  }
}
```

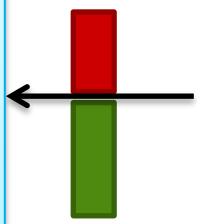
- **The value of `state` in iteration 1 depends on the value from iteration 0**
- **Similarly, iteration 2 depends on the value from iteration 1, etc**

ALTERA
MEASURABLE ADVANTAGE™

Loop Carried Dependencies

- **To achieve acceleration, we can pipeline each iteration of a loop containing loop carried dependencies**
  - Analyze any dependencies between iterations
  - Schedule these operations
  - Launch the next iteration as soon as possible

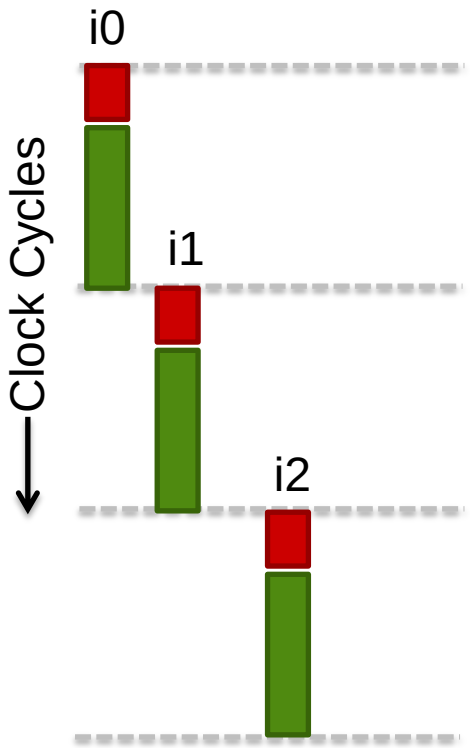
```
kernel void state_machine(ulong n)
{
  t_state_vector state = initial_state();
  for (ulong i=0; i<n; i++) {
    state = next_state( state );
    unit y = process( state );
    // more work…
  }
}
```



At this point, we can launch the next iteration

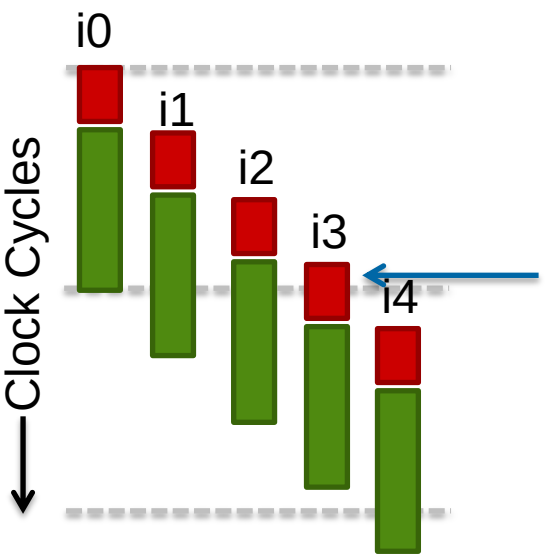
Loop Pipelining Example

- **No Loop Pipelining**



No Overlap of Iterations

- **With Loop Pipelining**



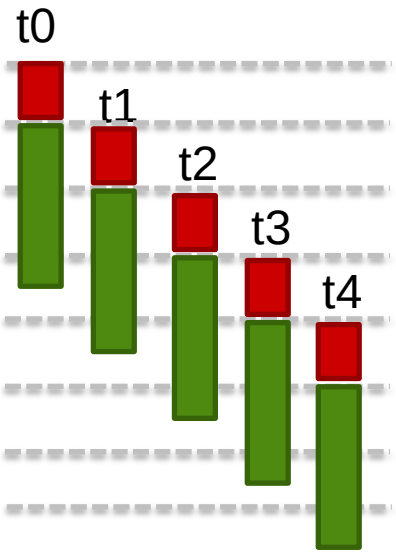
Looks almost like ND-range thread execution!

Finishes Faster because Iterations Are Overlapped

ALTERA
MEASURABLE ADVANTAGE™

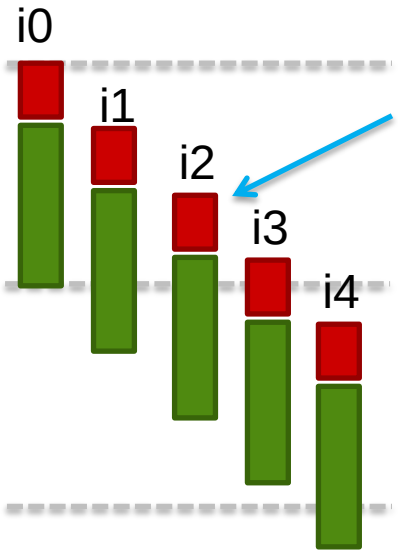
Pipelined Threads vs. Loop Pipelining

- **So what's the difference?**



Pipelined threads launch 1 thread per clock cycle in pipelined fashion

Pipelined Threads



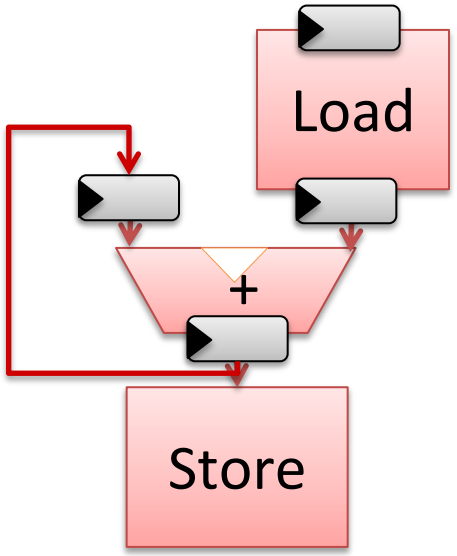
Loop dependencies may not be resolved in 1 clock cycle

Loop Pipelining

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

Accumulator Datapath

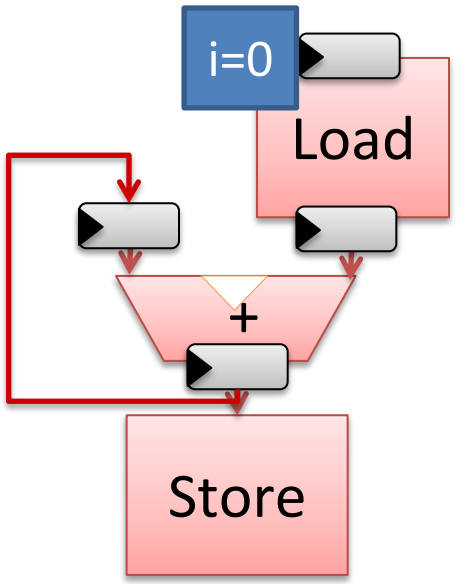
```
__kernel void
accumulate(__global float *a,
           __global float *b,
           int n)
{
  for (int i=1; i<n; ++i)
    b[i] = b[i-1] + a[i];
}
```



- A new iteration can be launched each cycle
- Each iteration still takes multiple cycles to complete, but subsequent iterations are not bottlenecked

Accumulator Datapath

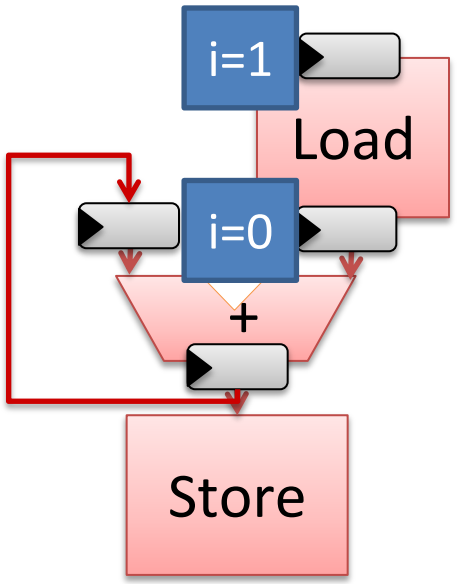
```
__kernel void
accumulate(__global float *a,
           __global float *b,
           int n)
{
  for (int i=1; i<n; ++i)
    b[i] = b[i-1] + a[i];
}
```



- A new iteration can be launched each cycle
- Each iteration still takes multiple cycles to complete, but subsequent iterations are bottlenecked

Accumulator Datapath

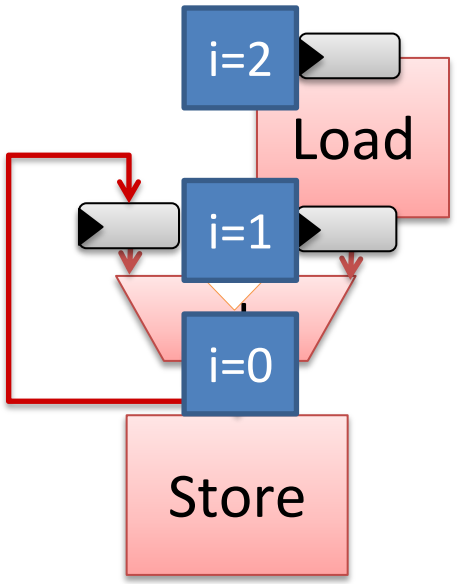
```
__kernel void
accumulate(__global float *a,
           __global float *b,
           int n)
{
  for (int i=1; i<n; ++i)
    b[i] = b[i-1] + a[i];
}
```



- A new iteration can be launched each cycle
- Each iteration still takes multiple cycles to complete, but subsequent iterations are bottlenecked

Accumulator Datapath

```
__kernel void
accumulate(__global float *a,
           __global float *b,
           int n)
{
  for (int i=1; i<n; ++i)
    b[i] = b[i-1] + a[i];
}
```



- A new iteration can be launched each cycle
- Each iteration still takes multiple cycles to complete, but subsequent iterations are bottlenecked

Dependence Analysis

- **Has profound effect on Loop Pipelining**
  - Can lead to difference in performance of more than 100x
- **Significant effort spent to improve dependence analysis**
  - Especially loop-carried dependence analysis
- **Added complex range analysis to help**
- **Uses knowledge of our specialized hardware and programming model**
- **Never good enough!**

ALTERA
MEASURABLE ADVANTAGE™

LLVM Issues/Wishlist

LLVM Issues

- **Intrinsics don't support structs**
  - We extended CallInst for our intrinsics
- **Module pass managers running every analysis on every function when only requesting a single function**
- **On-the-fly pass manager not inheriting analyses**
- **Ran into several scaling problems with LLVM passes**
  - Often due to significant loop unrolling and inlining
- **Loop representation**
  - Well formed loops are extremely important to us
  - Some optimizations introduce extra loops
  - `while(1)` with no return is useful to us

ALTERA
MEASURABLE ADVANTAGE™

LLVM Wishlist

- Conditional preservation of analyses
- Windows debug support
- Improved dependence analysis

Thank You

ALTERA®

MEASURABLE ADVANTAGE™

References

- **Altera OpenCL Example Designs**
  http://www.altera.com/support/examples/opencl/opencl.html
- **Altera OpenCL Best Practices Guide**
  http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf
- **Stratix V Overview**
  http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/stxv-index.jsp
- **Cyclone V Overview**
  http://www.altera.com/devices/fpga/cyclone-v-fpgas/cyv-index.jsp
- **Stratix V ALM**
  www.altera.com/literature/hb/stratix-v/stx5_51002.pdf

ALTERA
MEASURABLE ADVANTAGE™