Resource and Memory Management Techniques for the High-Level Synthesis of Software Threads into Parallel FPGA Hardware

Jongsok Choi, Stephen Brown, and Jason Anderson

FPT 2015
December 9, 2015
Queenstown, New Zealand

Dept. of Electrical and Computer Engineering
University of Toronto

Motivation

- High-level synthesis (HLS) can automatically generate hardware (HW) from software (SW)
  - A designer can work more productively at a higher level of abstraction, reducing time-to-market vs. hand-coded RTL
- Modern FPGAs are **very** large
  - Virtex UltraScale XCVU440 has 4.4 million logic cells, with more than 20 billion transistors, making it the world's densest IC
  - A designer needs to leverage the available resources **intelligently** to achieve the highest performance
  - With HLS, one can exploit FPGA's spatial parallelism much **easier** than when designing in RTL

Motivation

- LegUp HLS can automatically synthesize Pthreads into parallel-operating HW modules
- Each thread is synthesized into a concurrent HW instance
  - This approach enables SW engineers to leverage **HW parallelism** through a **SW programming paradigm** they are likely familiar with

- In the context of concurrently operating HW modules compiled from Pthreads:
  1. How should the different modules be connected together? (Circuit topology)
  2. How should the memories be shared between parallel threads? (Memory architecture)

LegUp HLS Framework

- Open-source HLS tool from the University of Toronto
  - Compiles C to Verilog
  - Leverages the LLVM (3.5) compiler infrastructure
  - Supports 3 different targets
    1. Entire program compiled to HW (Pure HW)
    2. Accelerate a portion of the program to HW (Hybrid flow)
       - Soft MIPS processor
       - Hard ARM processor (Altera DE-1 SoC)
  - Supports 3 different FPGA vendors
    - Altera, Xilinx, Lattice
  - Freely distributed to the research community (www.legup.org)
    - On 4th release (August 2015)
    - 3000+ downloads since 1st release (March 2011)

LegUp HLS Framework

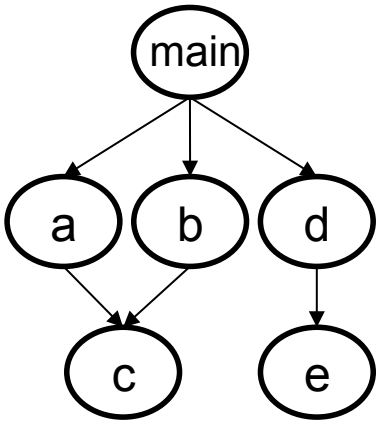
- Open-source HLS tool from the University of Toronto
  - Compiles C to Verilog
  - Leverages the LLVM (3.5) compiler infrastructure
  - Supports 3 different targets
    1. **Entire program compiled to HW (Pure HW)**
    2. Accelerate a portion of the program to HW (Hybrid flow)
       - Soft MIPS processor
       - Hard ARM processor (Altera DE-1 SoC)
  - Supports 3 different FPGA vendors
    - **Altera**, Xilinx, Lattice
  - Freely distributed to the research community (www.legup.org)
    - On 4th release (August 2015)
    - 3000+ downloads since 1st release (March 2011)

Circuit Topology

- Unlike software compilers, which target **fixed** processor architectures, HLS compilers can **optimize** a design's architecture for a specific application

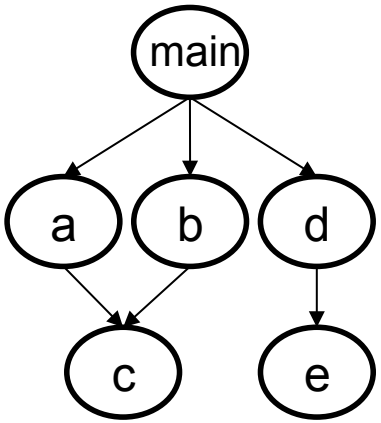
- Investigate two circuit topologies:
  1. Nested topology: HW module is self-contained, any other modules it uses are instantiated **inside** the module (default architecture used by LegUp, Vivado HLS)
  2. Flat topology: All HW modules are instantiated in a **common** module, different modules are connected through an interconnect

Circuit Topology: Nested

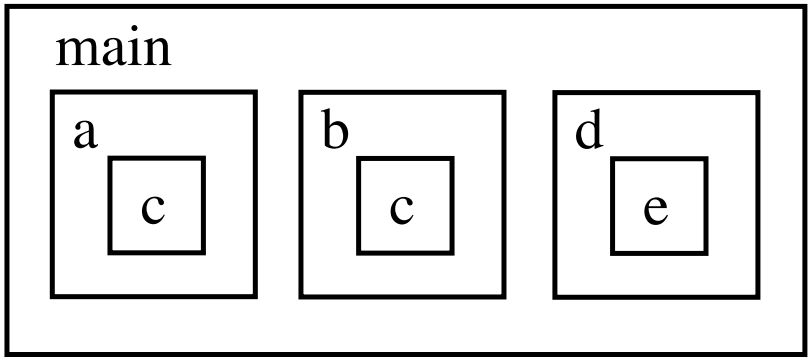


Program call graph

Circuit Topology: Nested

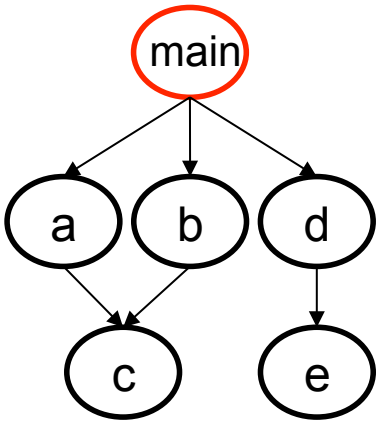


Program call graph

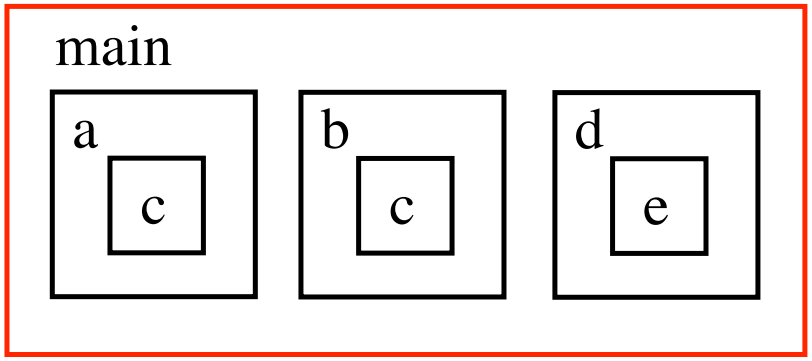


Circuit architecture

Circuit Topology: Nested

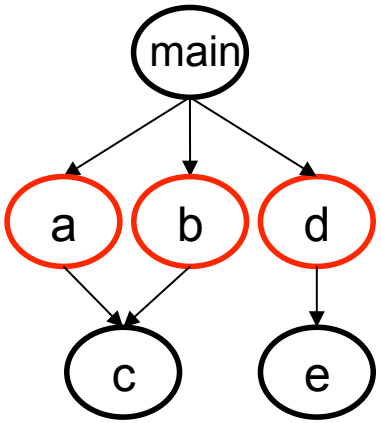


Program call graph

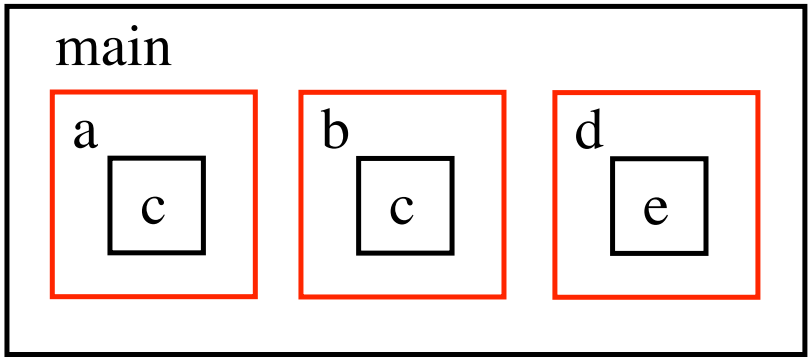


Circuit architecture

Circuit Topology: Nested

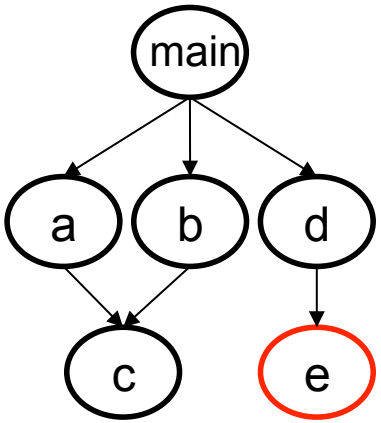


Program call graph

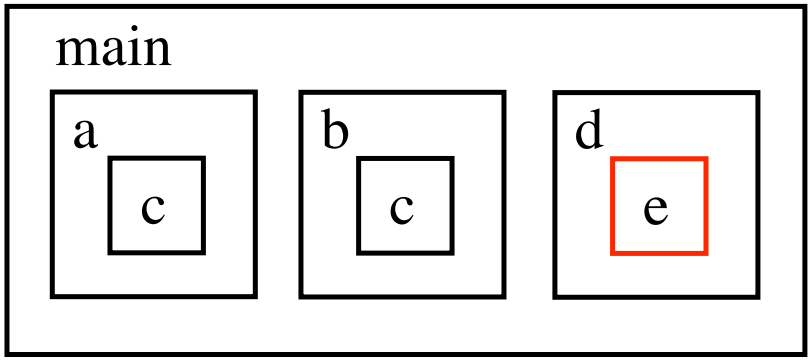


Circuit architecture

Circuit Topology: Nested

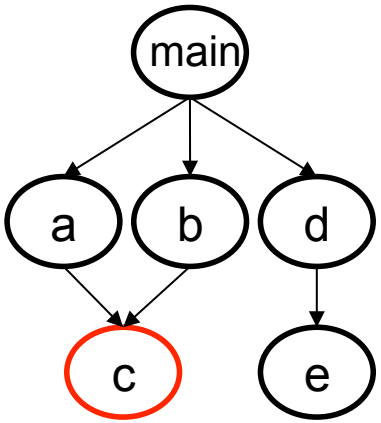


Program call graph

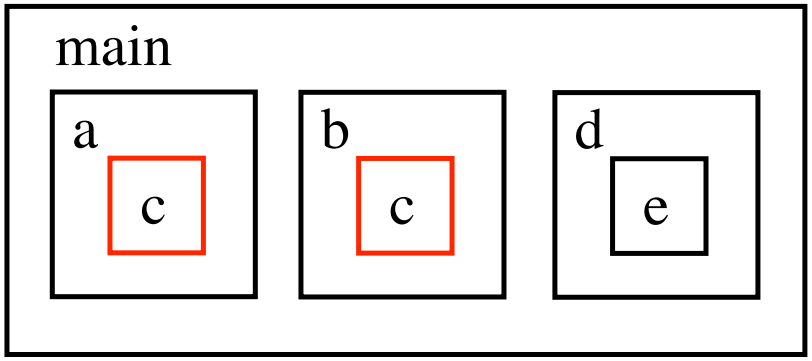


Circuit architecture

Circuit Topology: Nested

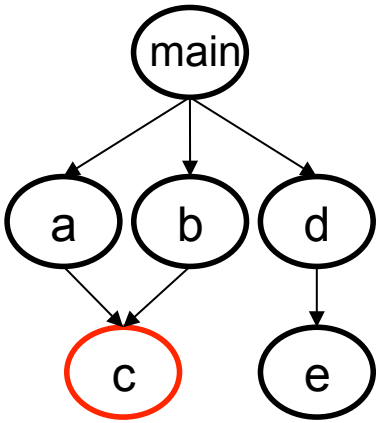


Program call graph

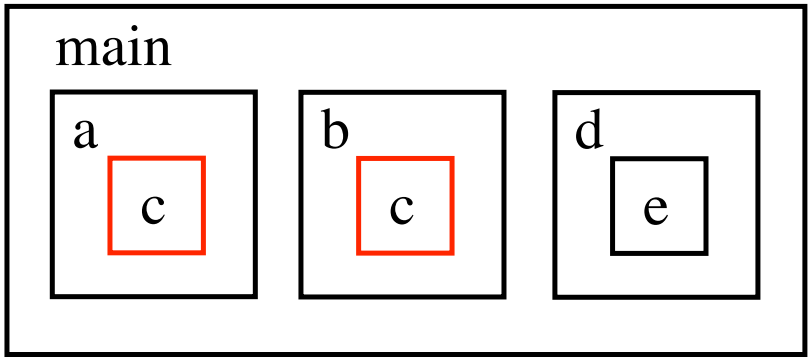


Circuit architecture

Circuit Topology: Nested



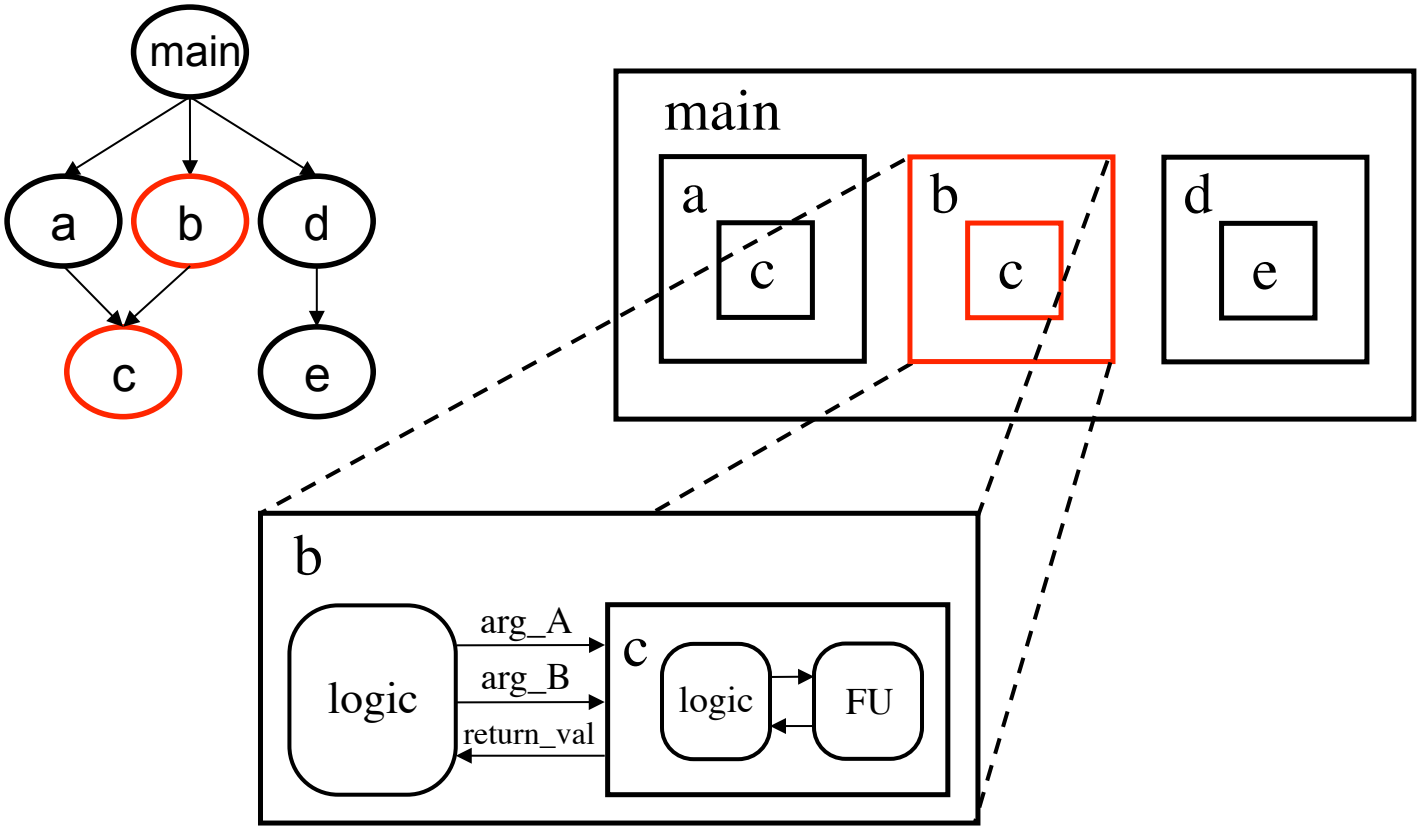
Program call graph



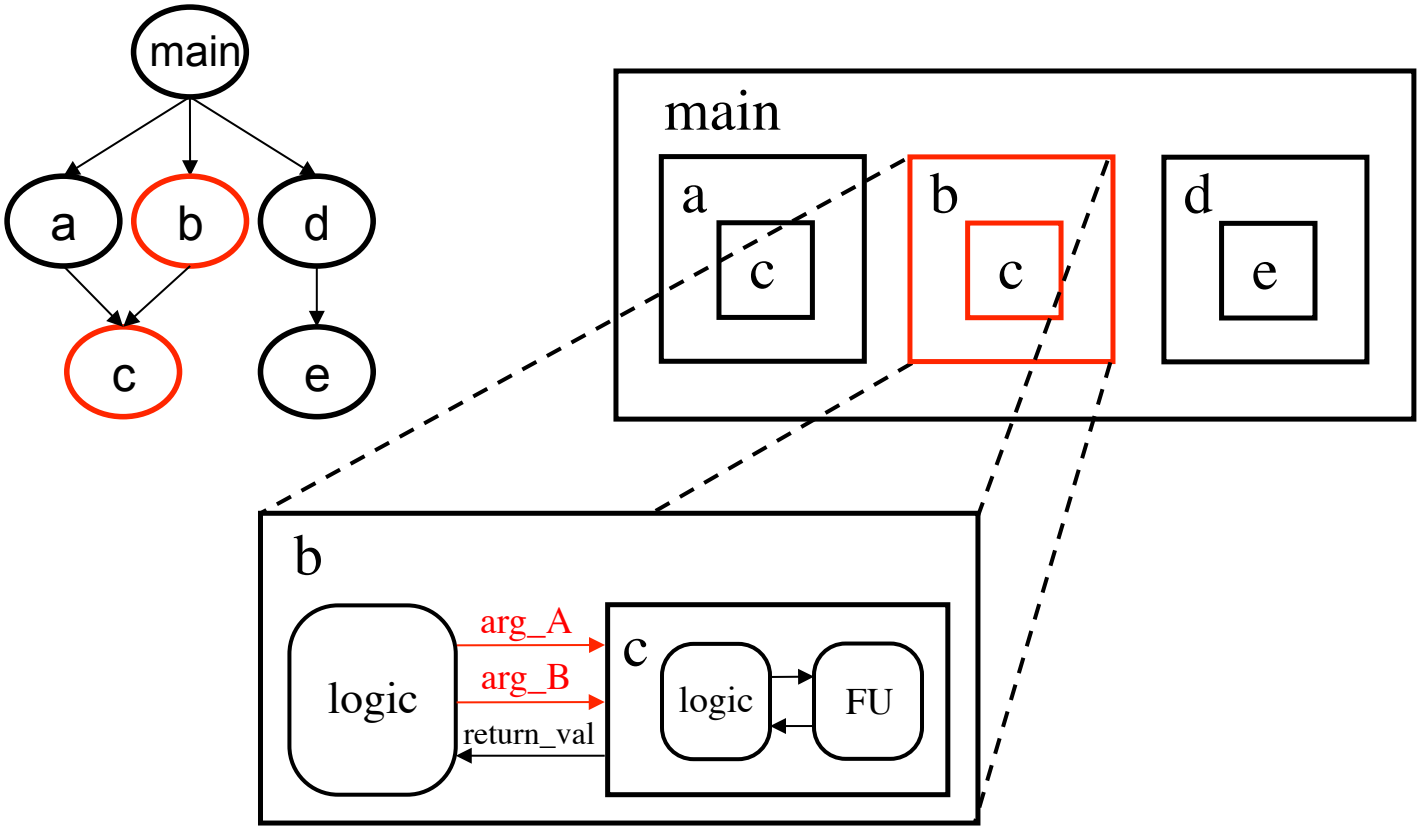
Circuit architecture

Module C is instantiate twice!

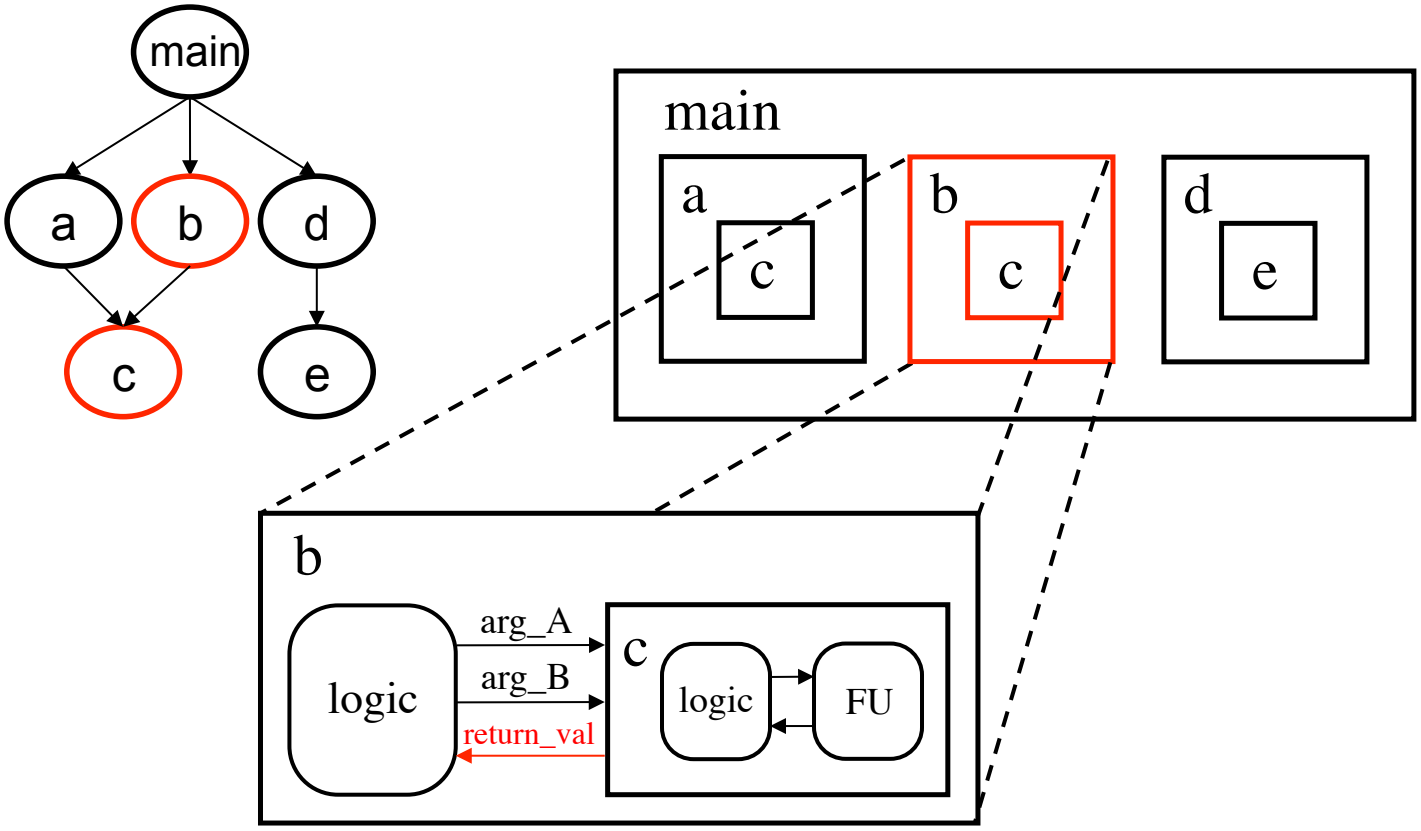
Circuit Topology: Nested



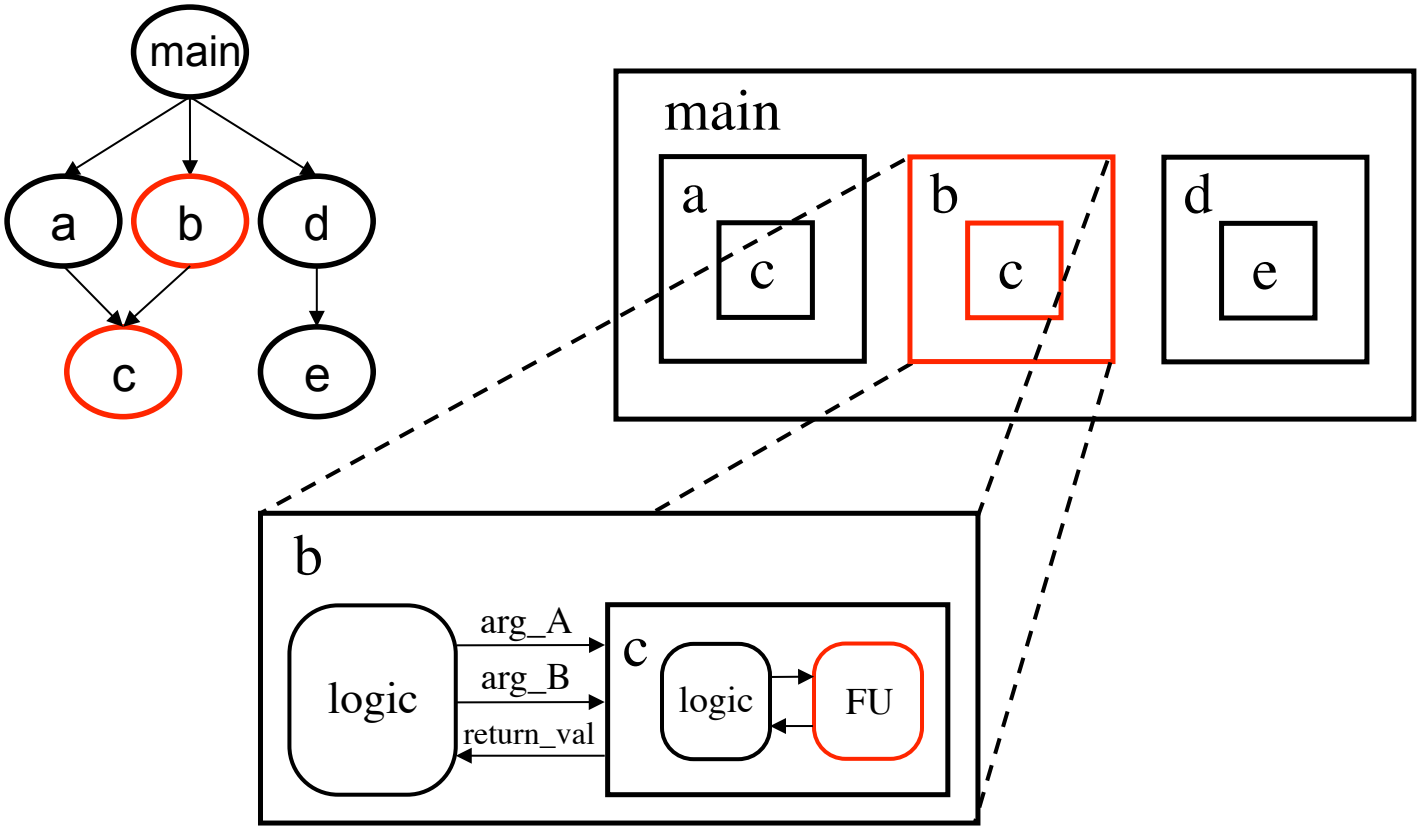
Circuit Topology: Nested



Circuit Topology: Nested



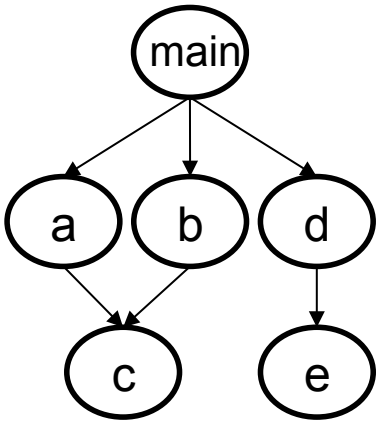
Circuit Topology: Nested



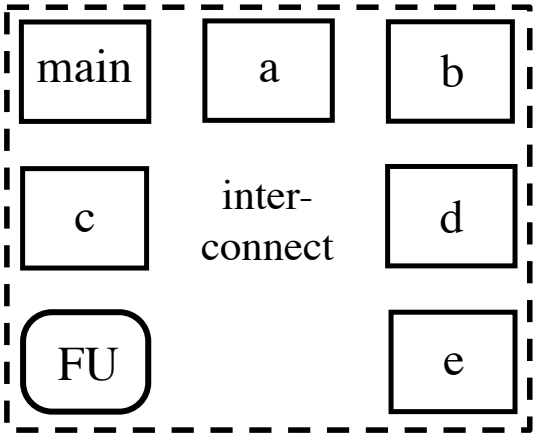
Circuit Topology: Nested

- Pros:
  - **Simplicity**: Any used modules are instantiated within the module itself and directly connected inside
  - The HW module interface is aligned to that of software
- Cons:
  - Can unnecessarily **replicate HW**
  - Functional units are instantiated within HW modules: **Sharing between modules is precluded**!

Circuit Topology: Flat



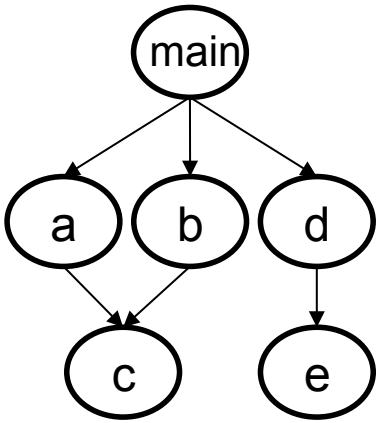
Program call graph



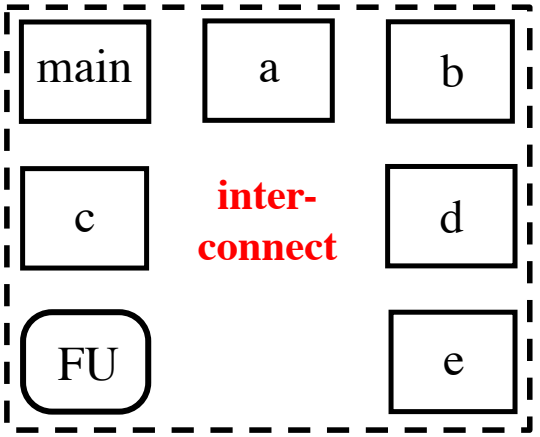
Circuit architecture

All modules reside in a common module

Circuit Topology: Flat

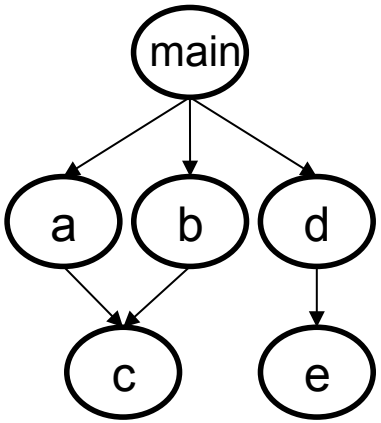


Program call graph

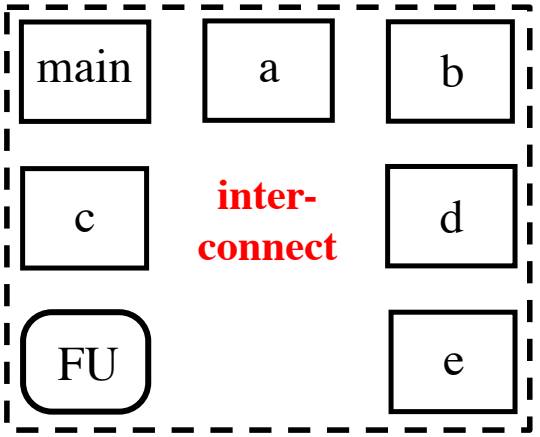


Circuit architecture

Circuit Topology: Flat



Program call graph



Circuit architecture

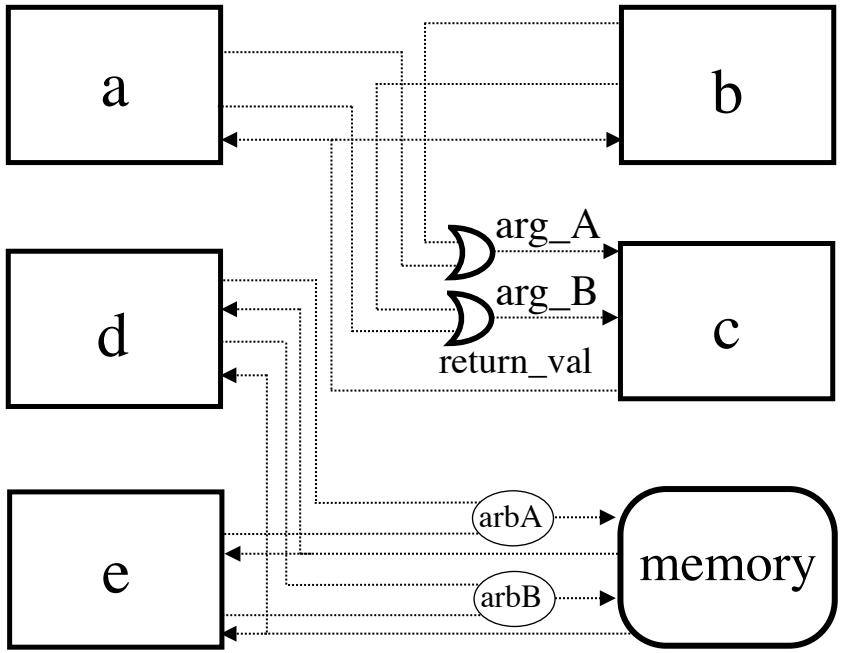
Interconnect is automatically generated by our System Generator

System Generator

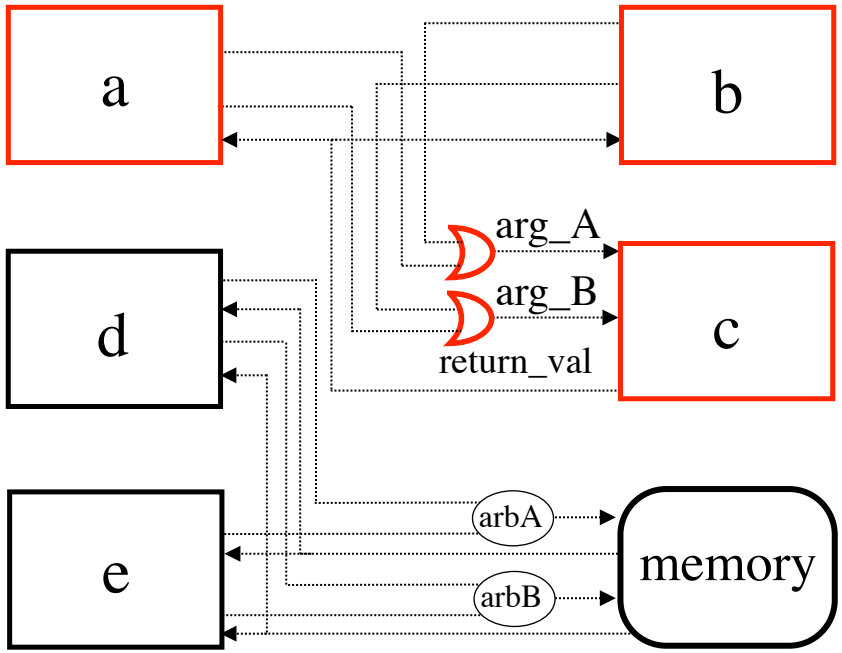
- Automatically connects all HW components by traversing the call graph of the input SW
  - Completely integrated into the HLS framework, requires no additional input from the user

- Handles both sequential and parallel execution differently
  - When multiple HW modules share a HW module, arbitration is automatically created
  - In sequential execution: a simple **OR gate** is created
  - In parallel execution: a **round-robin arbiter** is created

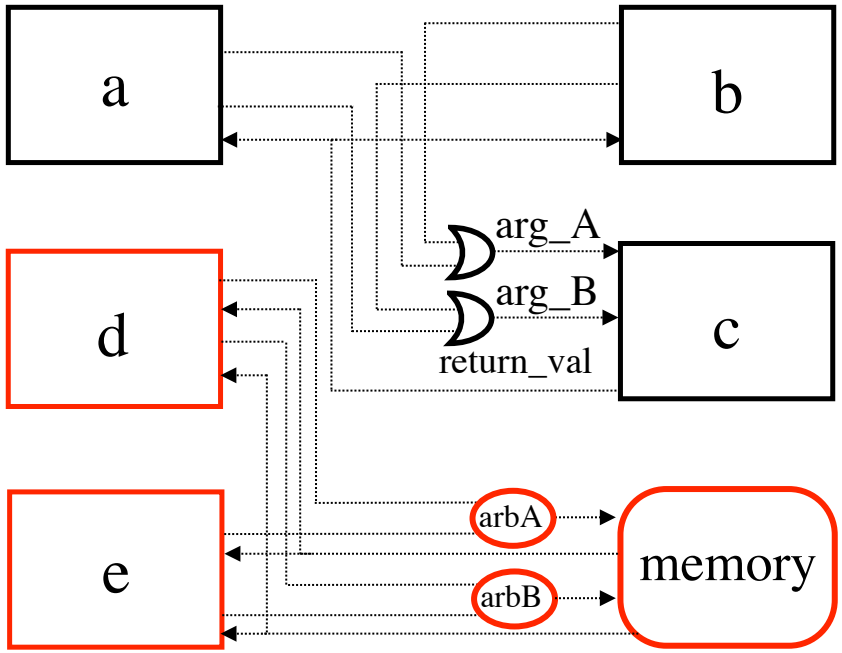
System Generator: Interconnect



System Generator: Interconnect

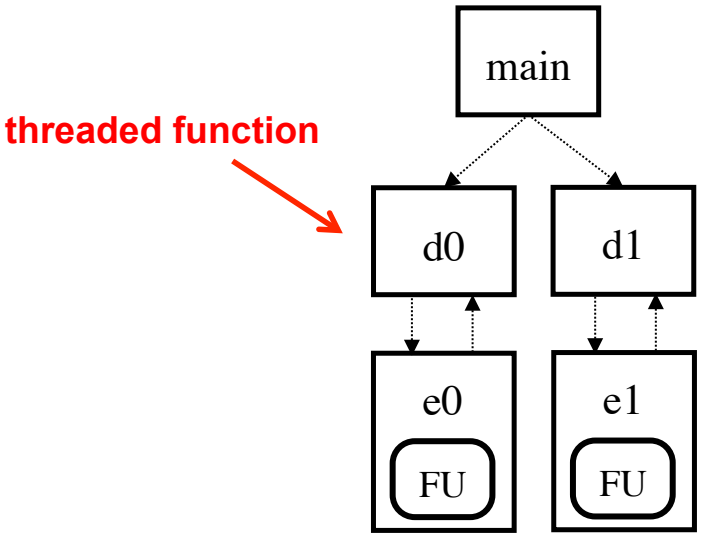


System Generator: Interconnect

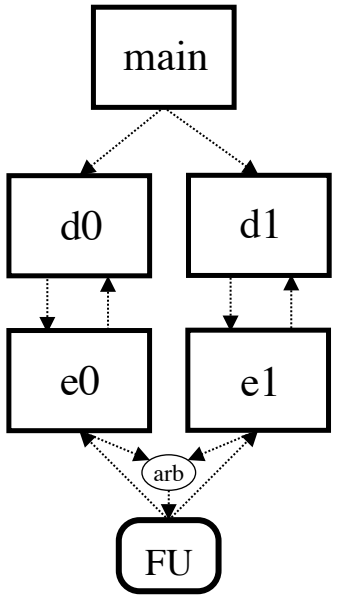


System Generator: Share or Replicate

- Can share or replicate HW modules based on input program and user constraints
  - For a threaded program: creates as many HW instances of the threaded function (+ its descendants) as the # of threads in SW
  - Allows sharing of functional units between parallel modules



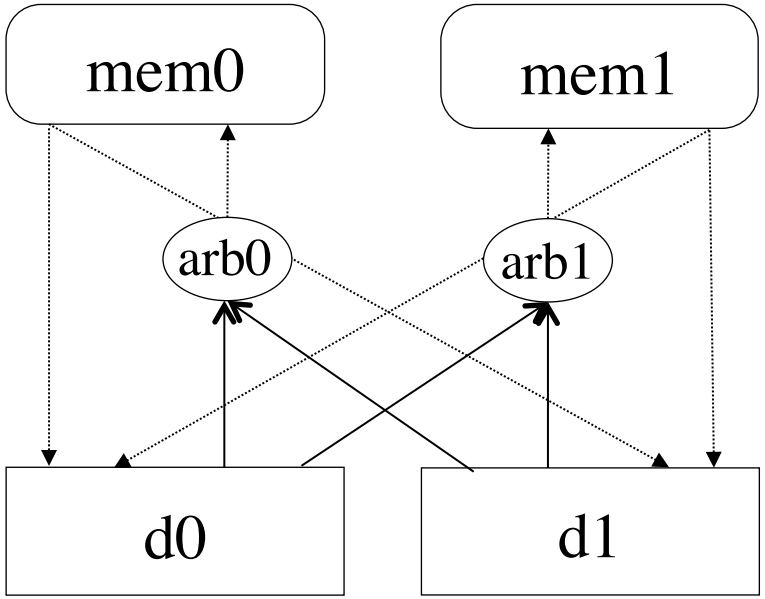
Parallel hardware **without** FU sharing



Parallel hardware **with** FU sharing

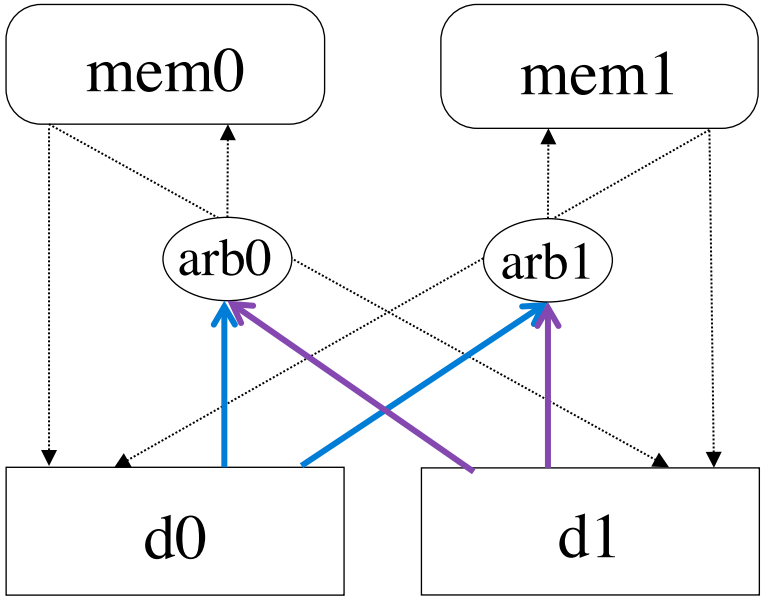
System Generator: Deadlock Prevention

- When multiple modules try to access multiple ***common*** modules at the same time, a ***deadlock*** can occur



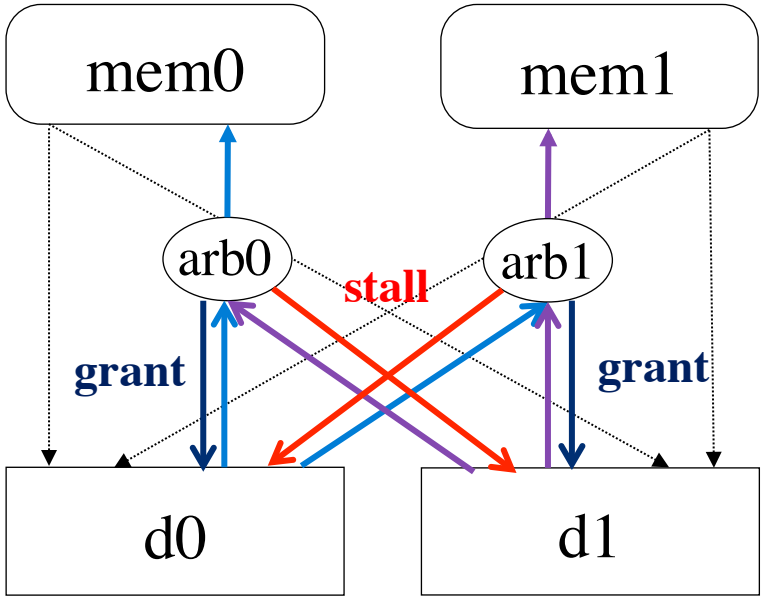
System Generator: Deadlock Prevention

- When multiple modules try to access multiple ***common*** modules at the same time, a ***deadlock*** can occur



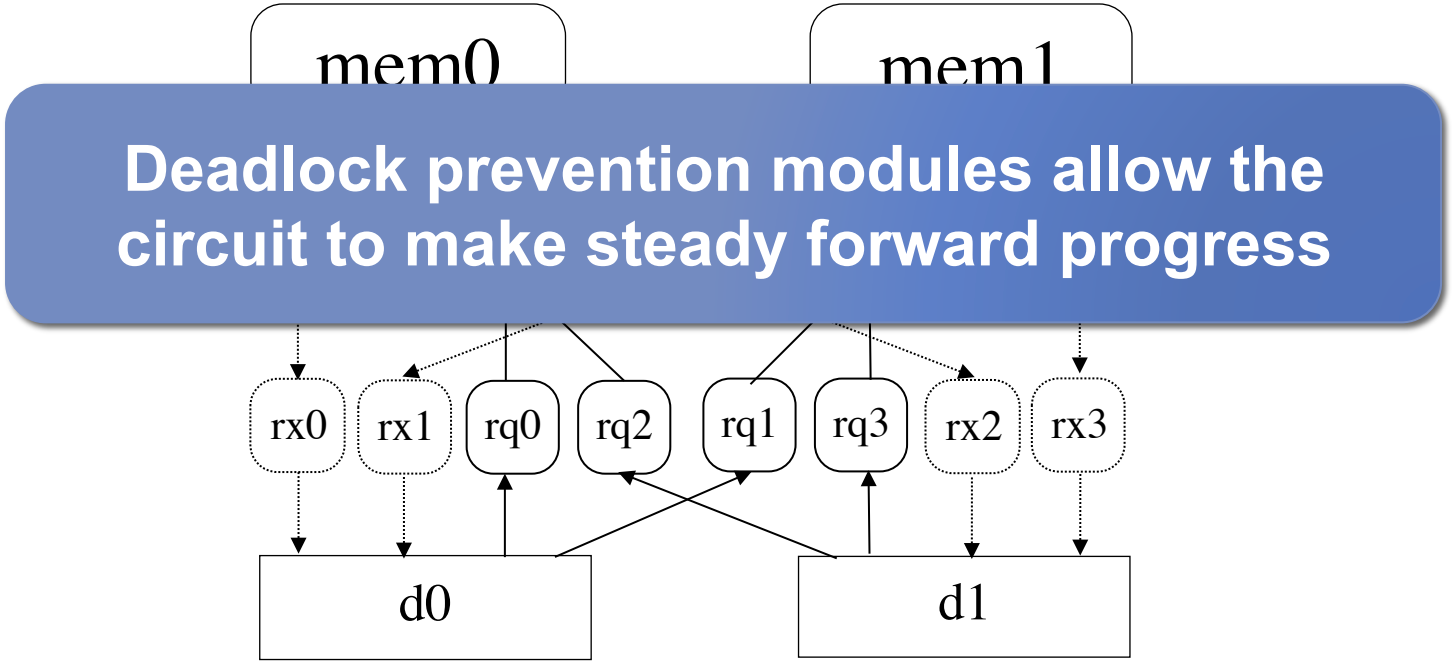
System Generator: Deadlock Prevention

- When multiple modules try to access multiple ***common*** modules at the same time, a ***deadlock*** can occur

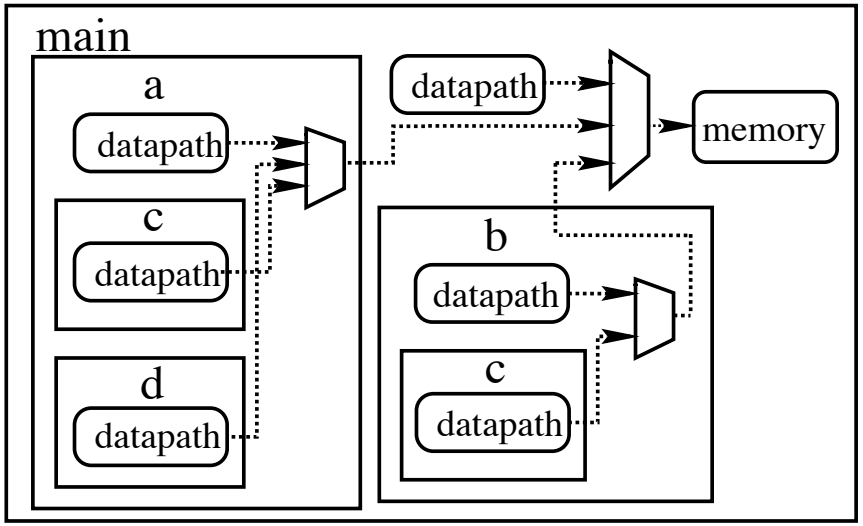


System Generator: Deadlock Prevention

- System generator automatically inserts deadlock prevention modules where necessary

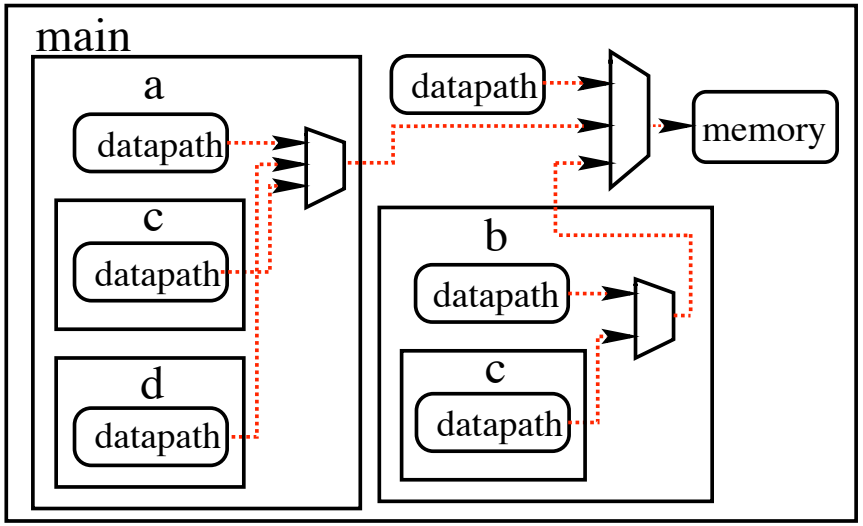


Competitive Edge: Flat Topology w/ System Generator



Nested topology
(sequential execution)

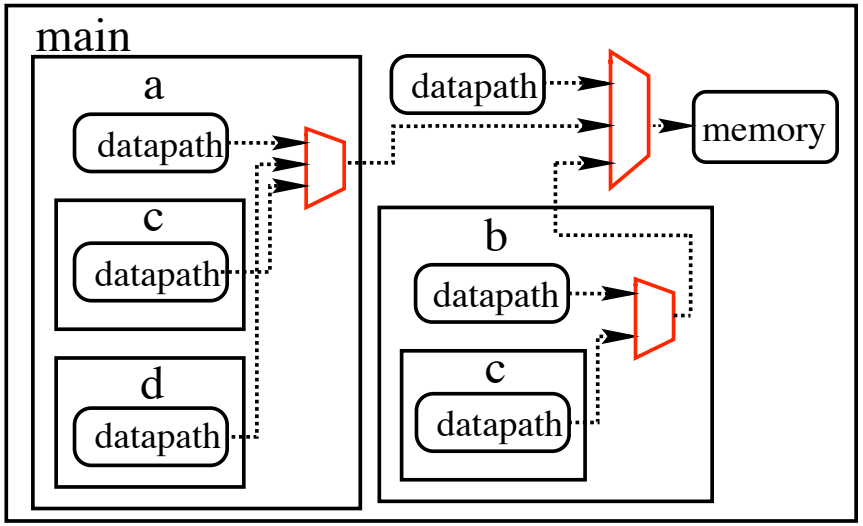
Competitive Edge: Flat Topology w/ System Generator



Nested topology
(sequential execution)

- Memory ports must be forwarded through all intermediate modules to memory

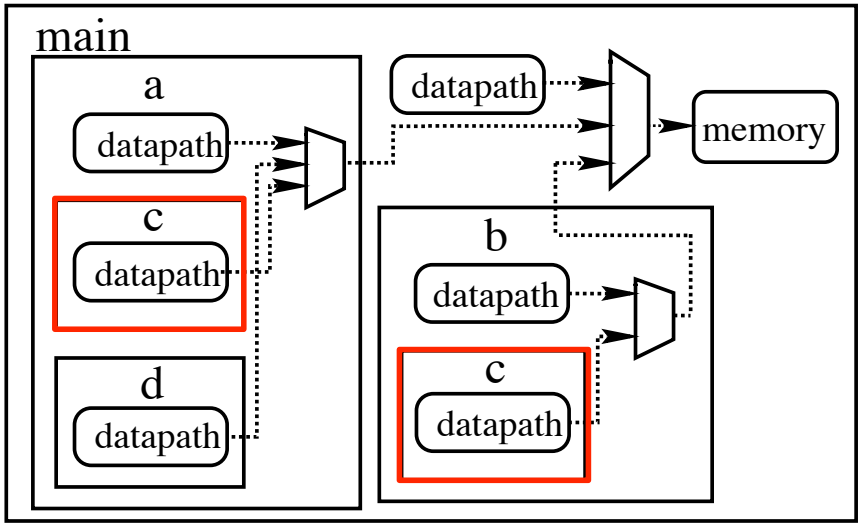
Competitive Edge: Flat Topology w/ System Generator



Nested topology
(sequential execution)

- Memory ports must be forwarded through all intermediate modules to memory
- Multiplexers are created at each level of hierarchy
  - The size/depth of the muxes grow with the # of functions that access the memory and the depth of the call hierarchy

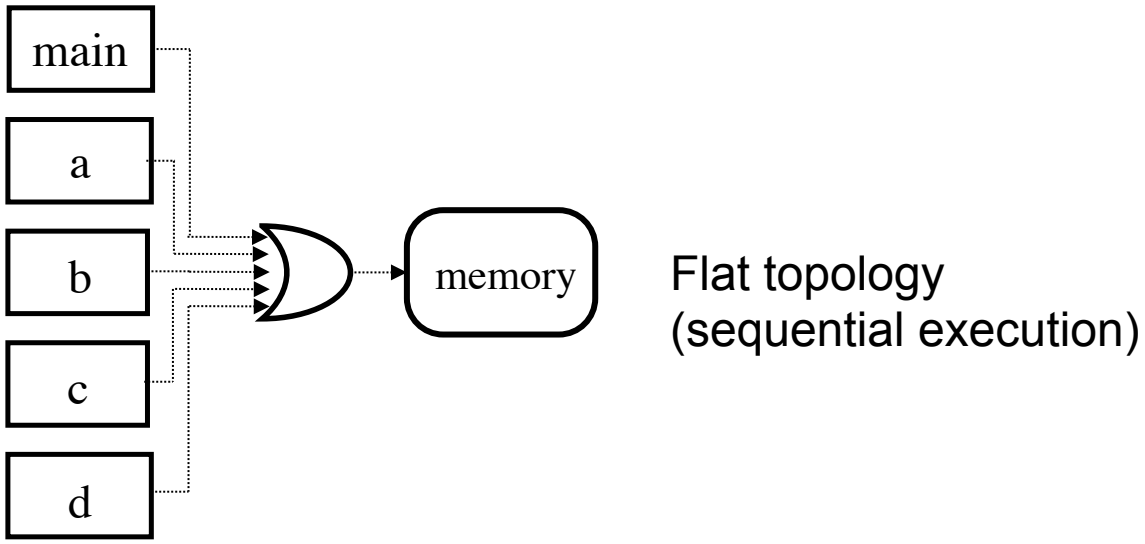
Competitive Edge: Flat Topology w/ System Generator



Nested topology (sequential execution)

- Memory ports must be forwarded through all intermediate modules to memory
- Multiplexers are created at each level of hierarchy
  - The size/depth of the muxes grow with the # of functions that access the memory and the depth of the call hierarchy
- Modules instantiated multiple times due to the nested topology also increase the multiplexer size unnecessarily

Competitive Edge: Flat Topology w/ System Generator



- No module replication
- No port forwarding
- Simple OR gate

Memory Architecture

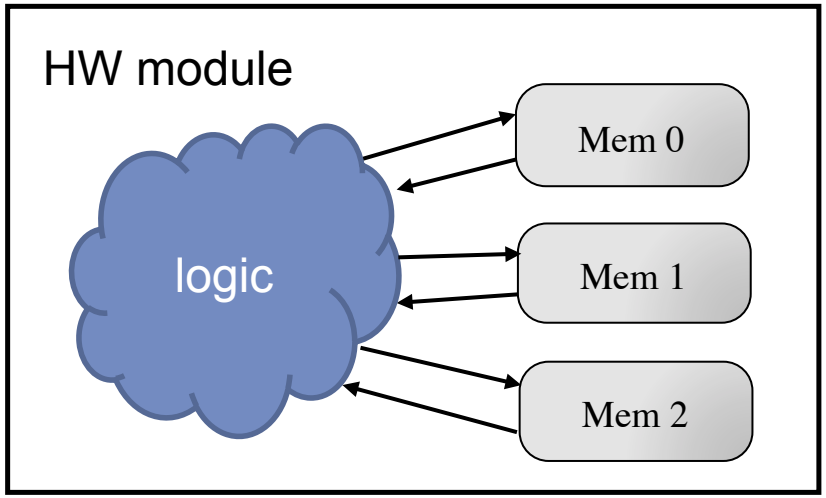
- Memory architecture plays a critical role in HW systems
  - Memory bandwidth can often be the limiting factor for performance
- FPGA characteristics:
  - Many on-chip block RAMs
  - Abundant registers
  - Both are distributed throughout the chip, and can be accessed in parallel with low latencies

- We want to make use of block RAMs and registers efficiently to increase memory bandwidth and reduce memory contention among parallel threads

Memory Architecture: Memory Partitioning

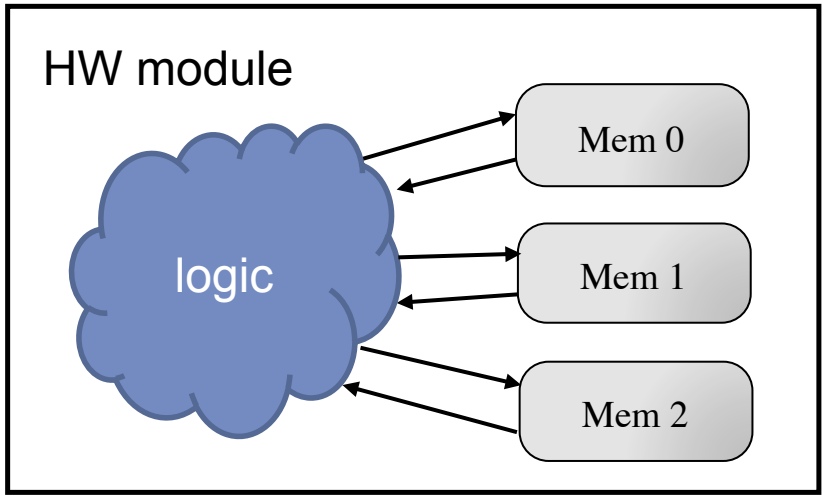
- Use points-to analysis to partition memories into **global**, **local**, and **shared-local** memories
  1. A pointer points to a **single** array:
     - Array is used by a single function – local memory
     - Array is used by multiple functions – shared-local memory
  2. A pointer points to **multiple** arrays:
     - Need to resolve pointer references at runtime – global memory

Memory Architecture: Local



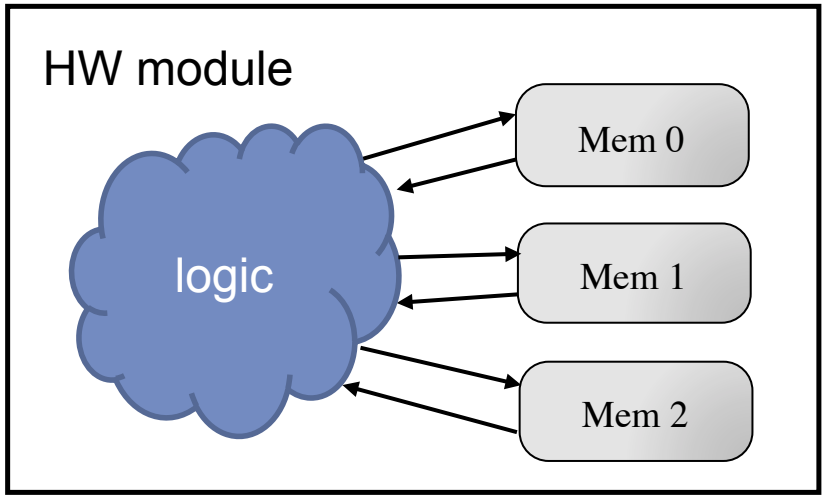
- Directly connected and instantiated inside the accessing module

Memory Architecture: Local



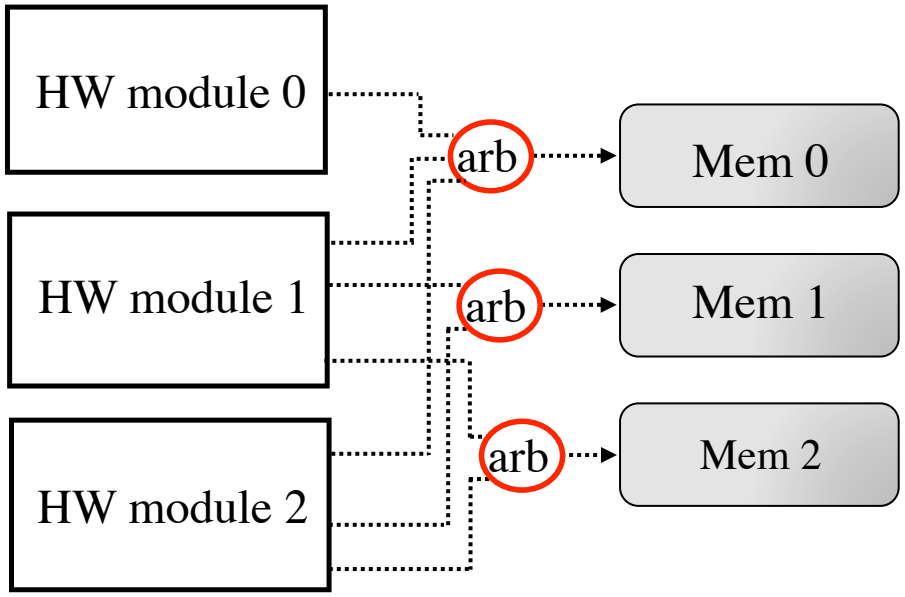
- Directly connected and instantiated inside the accessing module
- No arbitration required

Memory Architecture: Local



- Directly connected and instantiated inside the accessing module
- No arbitration required
- All local memories can be accessed concurrently

Memory Architecture: Shared-local



- Connected through arbitration logic (OR gate/arbiter), instantiated outside the accessing modules
- All independent shared-local memories can be accessed concurrently

Memory Architecture: Optimizations

- For local/shared-local memories:

Memory Architecture: Optimizations

- For local/shared-local memories:

1. Replicate read-only memories: localize a memory to its accessing module
   - Turns a shared-local memory to local memory
   - Eliminates stalls due to memory contention – **Higher performance**
   - Eliminates arbitration logic – **Lower LUT usage**

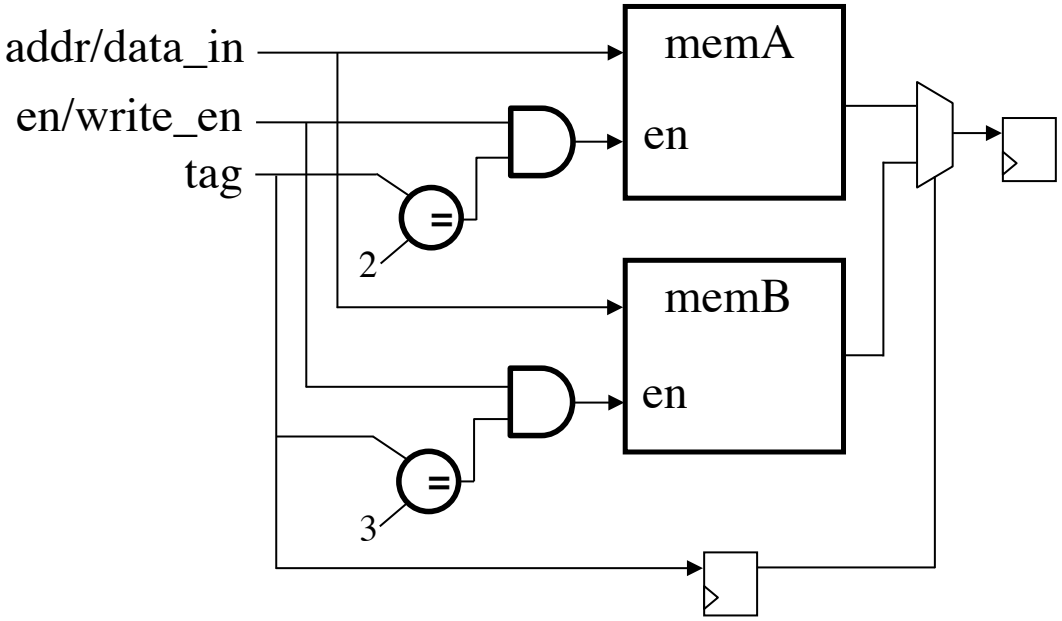
Memory Architecture: Optimizations

- For local/shared-local memories:

1. Replicate read-only memories: localize a memory to its accessing module
   - Turns a shared-local memory to local memory
   - Eliminates stalls due to memory contention – **Higher performance**
   - Eliminates arbitration logic – **Lower LUT usage**
2. Memory to register conversion: implement small arrays in registers
   - Reduces # of RAMs **– Lower RAM usage**
   - Reduces latency (RAMs have 1-cycle read latency, registers have 0-cycle read latency) – **Higher performance**

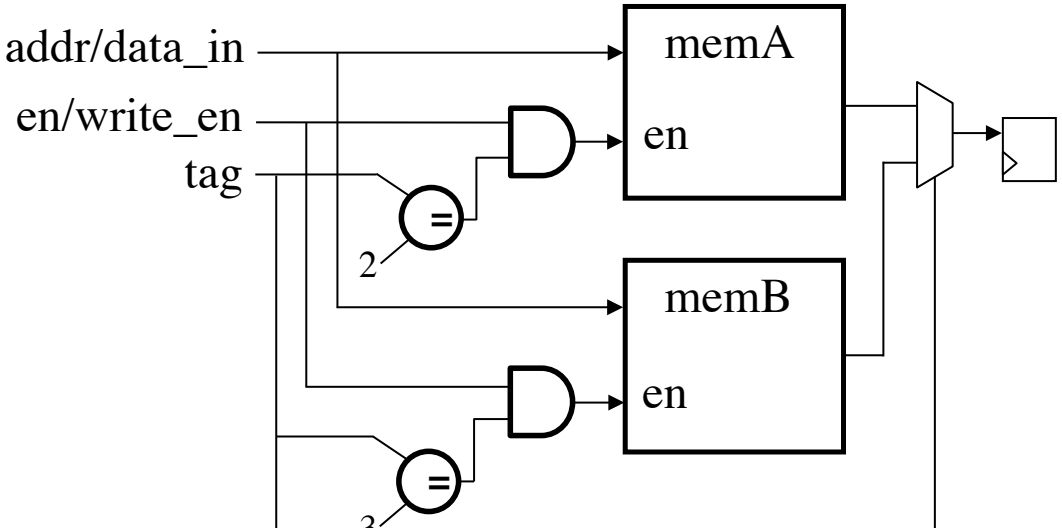
Memory Architectures: Global

- Global memories are created inside the global memory controller
  - Provides steering logic to steer memory access to the correct RAM at run-time



Memory Architectures: Global

- Global memories are created inside the global memory controller
  - Provides steering logic to steer memory access to the correct RAM at run-time



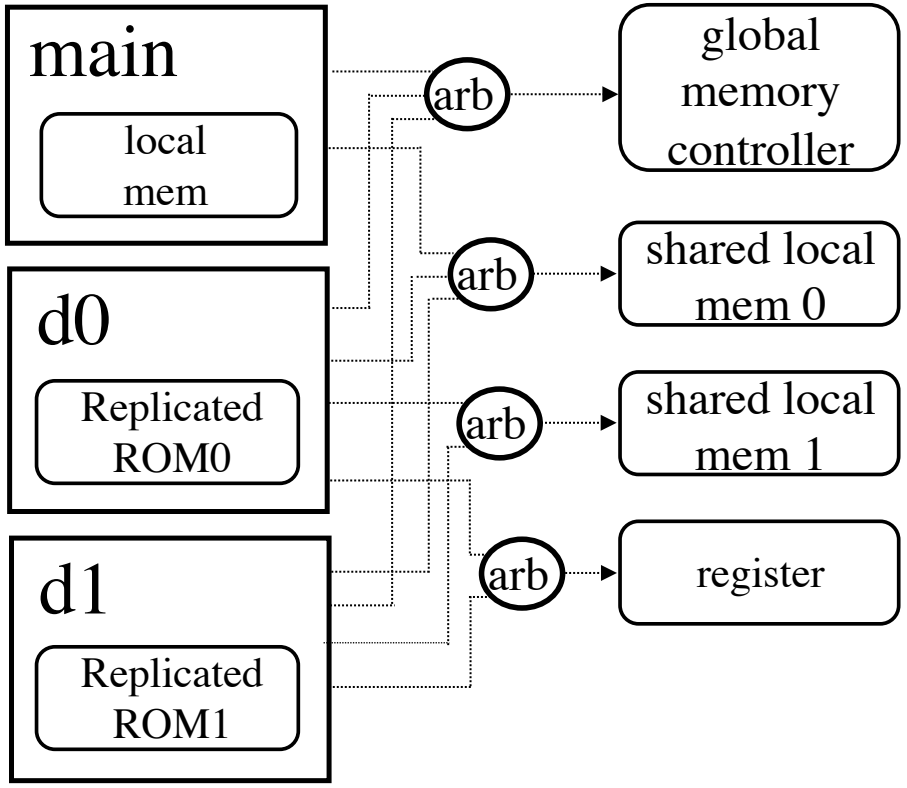
- Limits memory accesses to 2/cycle (RAMs are dual-port)
- Only created if necessary

Memory Architectures: Latencies

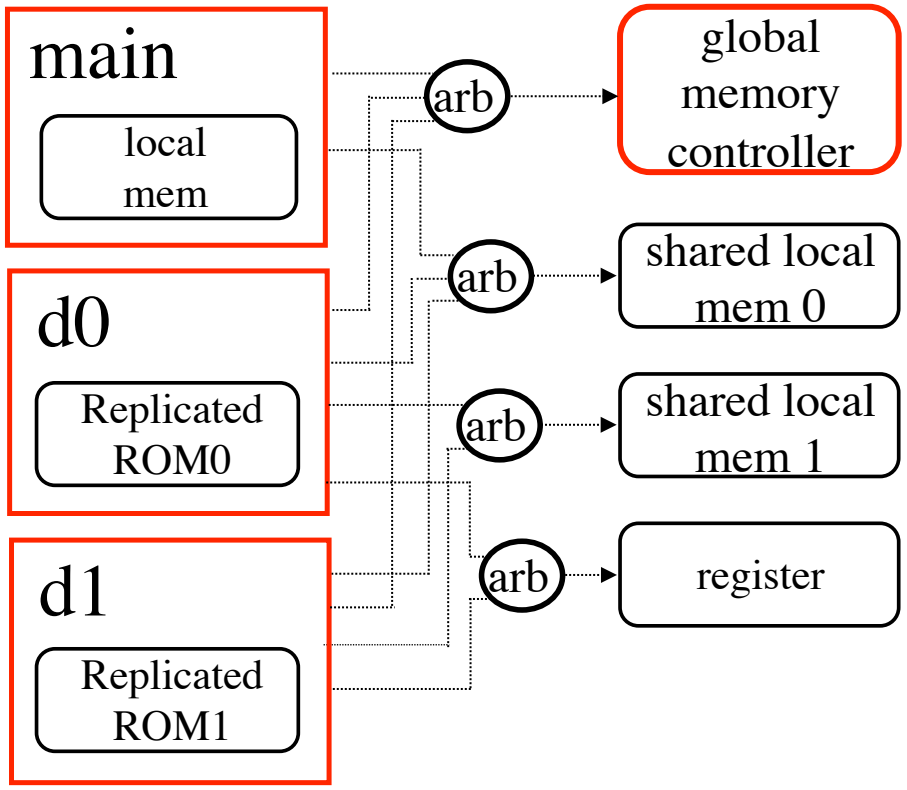
- We vary the *read* latencies for different types of memories

1. Global memories: **2-cycles**
   - Can be connected to many modules (high fan-in/fan-out)
   - Need to go through steering logic (can be large multiplexors)
2. Local/shared-local memories: **1-cycle**
   - Connected to a limited number of modules
   - Simple connections
3. Memories converted to registers: **0-cycle**

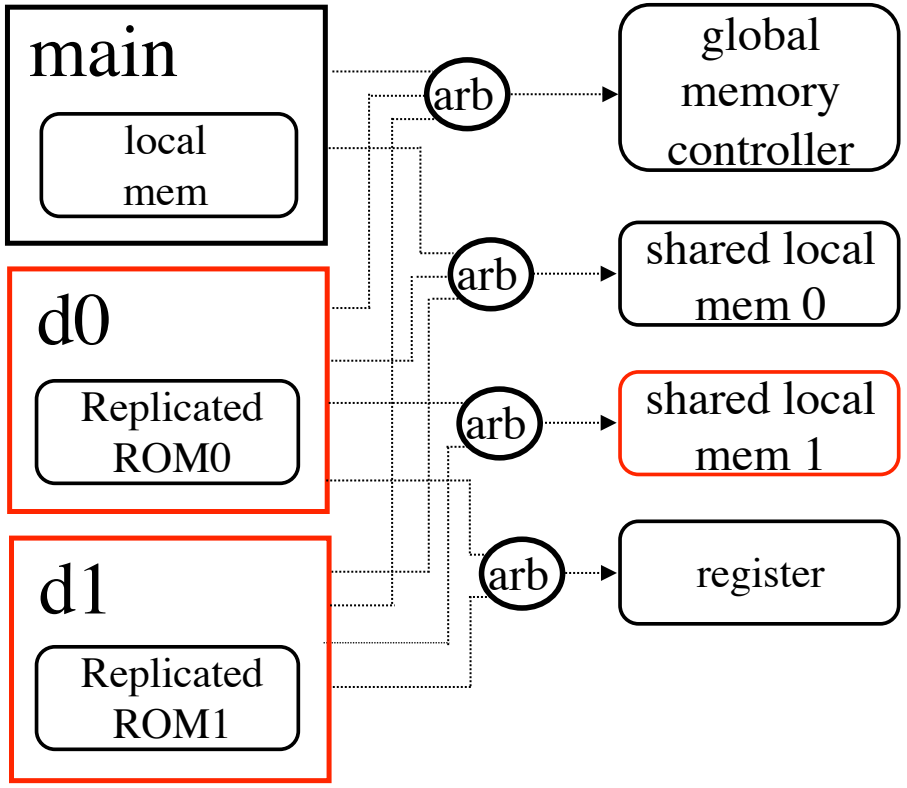
Memory Architecture: Example System



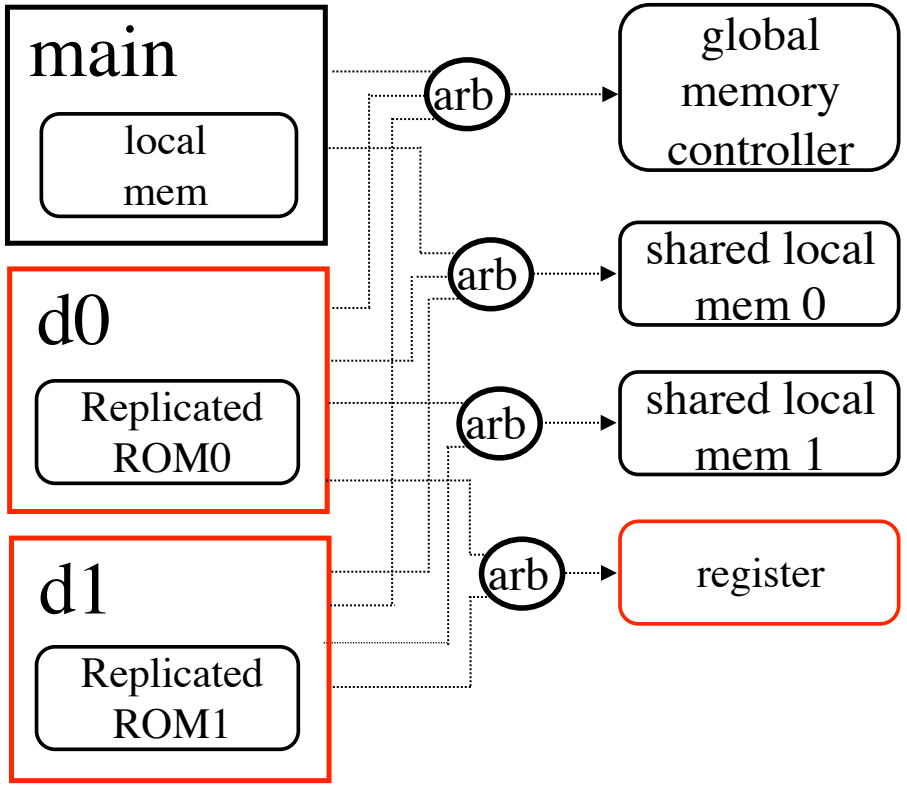
Memory Architecture: Example System



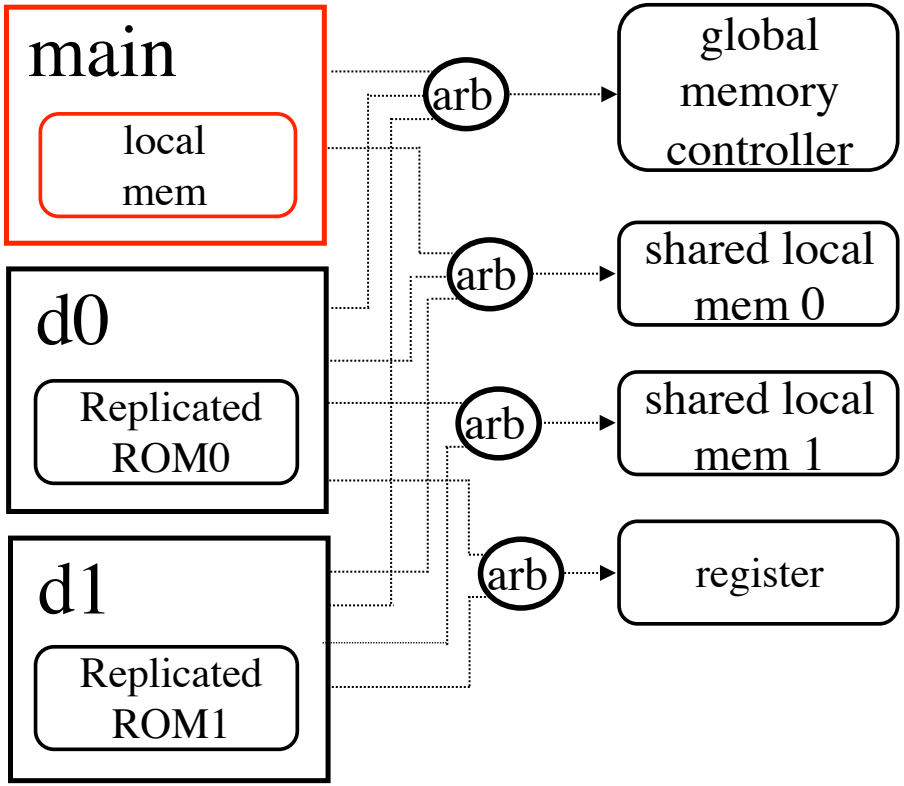
Memory Architecture: Example System



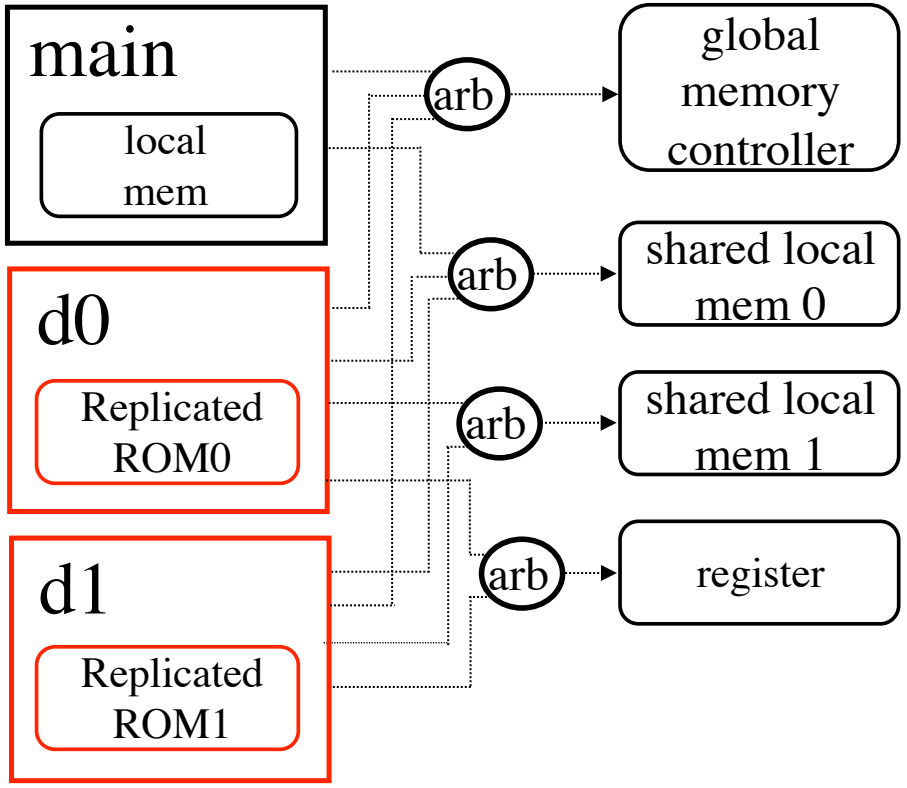
Memory Architecture: Example System



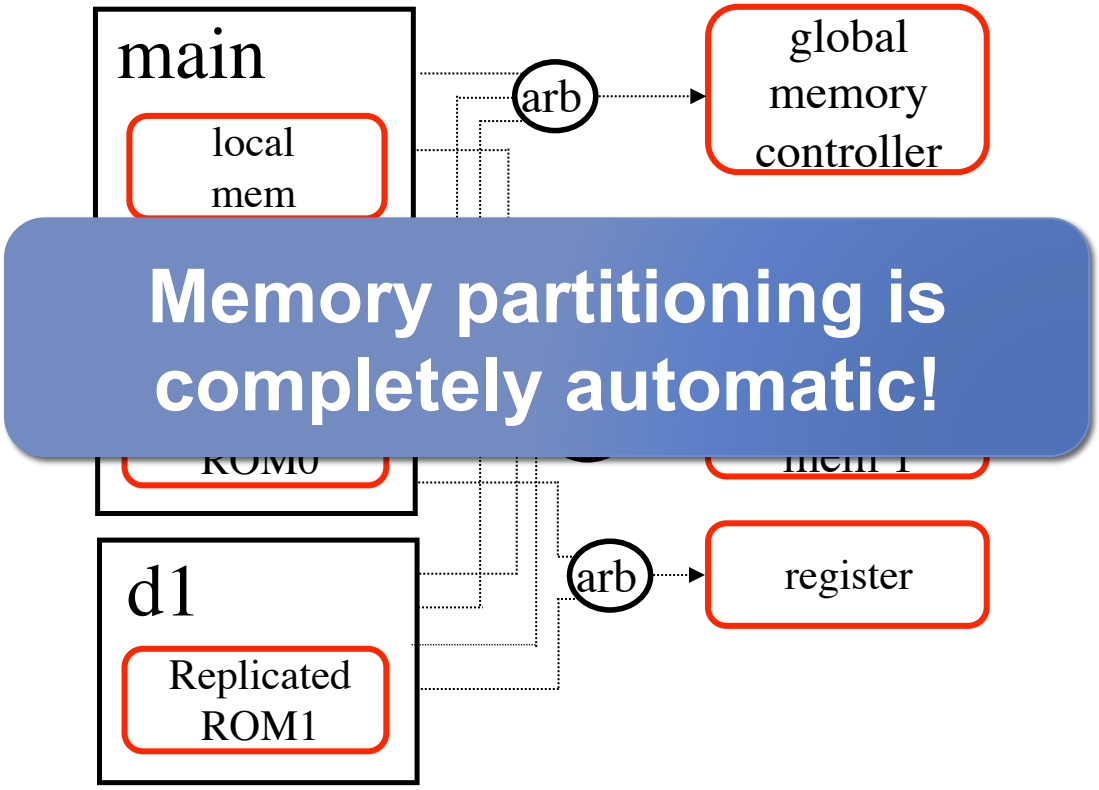
Memory Architecture: Example System



Memory Architecture: Example System



Memory Architecture: Example System



All independent memories can be accessed concurrently

Experimental Study

- We study the impact of different circuit/memory architectures on the performance/area of parallel HW

Experimental Study

- We study the impact of different circuit/memory architectures on the performance/area of parallel HW
- 8 different architectures:
  1. Nested topology with a global memory controller (baseline)

Experimental Study

- We study the impact of different circuit/memory architectures on the performance/area of parallel HW
- 8 different architectures:
  1. Nested topology with a global memory controller (baseline)
  2. Flat topology with a global memory controller

Experimental Study

- We study the impact of different circuit/memory architectures on the performance/area of parallel HW
- 8 different architectures:
  1. Nested topology with a global memory controller (baseline)
  2. Flat topology with a global memory controller
  3. Arch. 2 + divider sharing

Experimental Study

- We study the impact of different circuit/memory architectures on the performance/area of parallel HW
- 8 different architectures:
  1. Nested topology with a global memory controller (baseline)
  2. Flat topology with a global memory controller
  3. Arch. 2 + divider sharing
  4. Arch. 3 + plus multiplier sharing

Experimental Study

- We study the impact of different circuit/memory architectures on the performance/area of parallel HW
- 8 different architectures:
  1. Nested topology with a global memory controller (baseline)
  2. Flat topology with a global memory controller
  3. Arch. 2 + divider sharing
  4. Arch. 3 + plus multiplier sharing
  5. Arch. 4 + local/shared-local memories (memory latency of 2 cycles)

Experimental Study

- We study the impact of different circuit/memory architectures on the performance/area of parallel HW
- 8 different architectures:
  1. Nested topology with a global memory controller (baseline)
  2. Flat topology with a global memory controller
  3. Arch. 2 + divider sharing
  4. Arch. 3 + plus multiplier sharing
  5. Arch. 4 + local/shared-local memories (memory latency of 2 cycles)
  6. Arch. 5 + memory to register conversion (local/shared-local memory latency = 1, register latency = 0)

Experimental Study

- We study the impact of different circuit/memory architectures on the performance/area of parallel HW
- 8 different architectures:
  1. Nested topology with a global memory controller (baseline)
  2. Flat topology with a global memory controller
  3. Arch. 2 + divider sharing
  4. Arch. 3 + plus multiplier sharing
  5. Arch. 4 + local/shared-local memories (memory latency of 2 cycles)
  6. Arch. 5 + memory to register conversion (local/shared-local memory latency = 1, register latency = 0)
  7. Arch. 6 + constant memory replication

Experimental Study

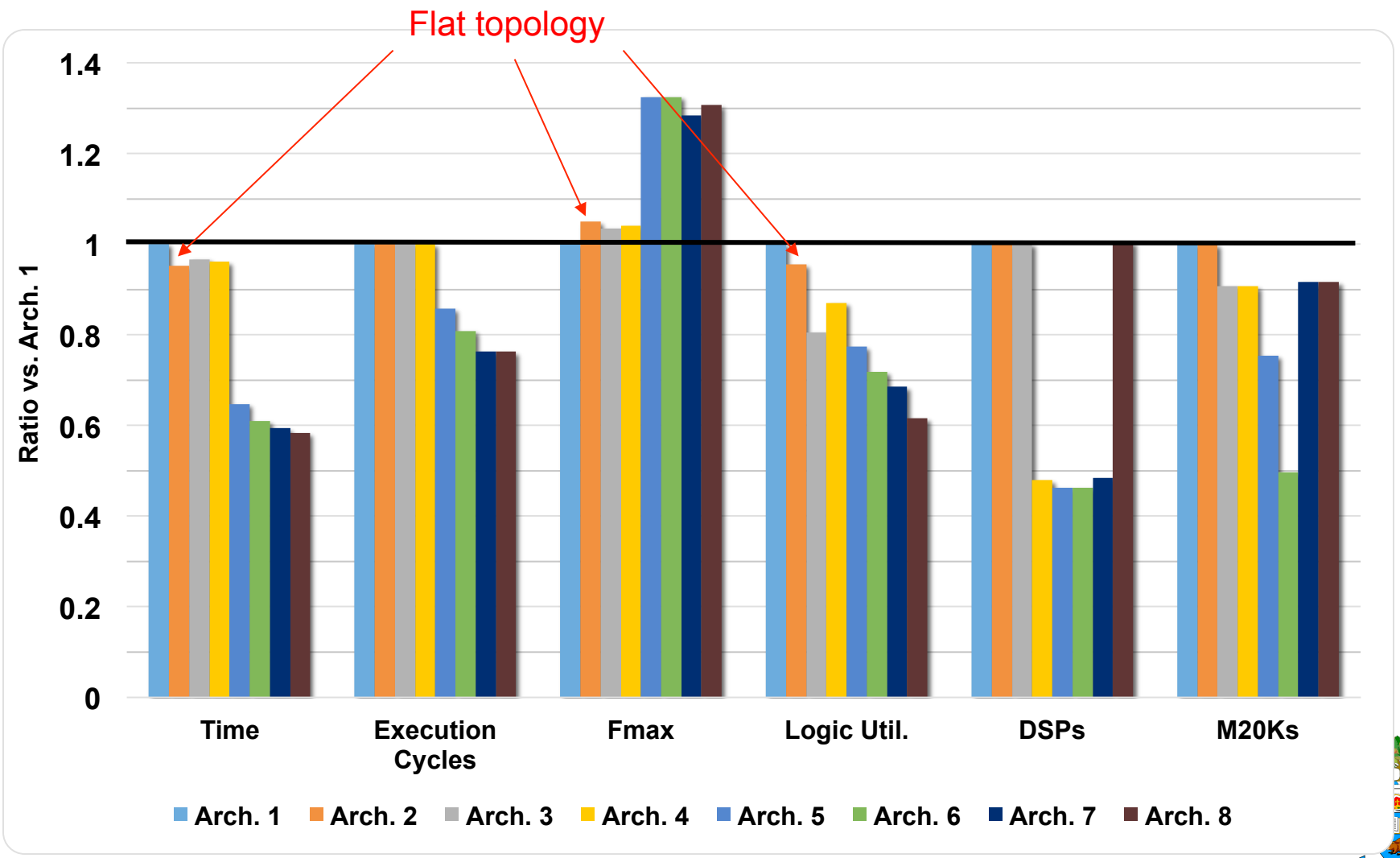
- We study the impact of different circuit/memory architectures on the performance/area of parallel HW
- 8 different architectures:
  1. Nested topology with a global memory controller (baseline)
  2. Flat topology with a global memory controller
  3. Arch. 2 + divider sharing
  4. Arch. 3 + plus multiplier sharing
  5. Arch. 4 + local/shared-local memories (memory latency of 2 cycles)
  6. Arch. 5 + memory to register conversion (local/shared-local memory latency = 1, register latency = 0)
  7. Arch. 6 + constant memory replication
  8. Arch. 7 - multiplier sharing

Experimental Study

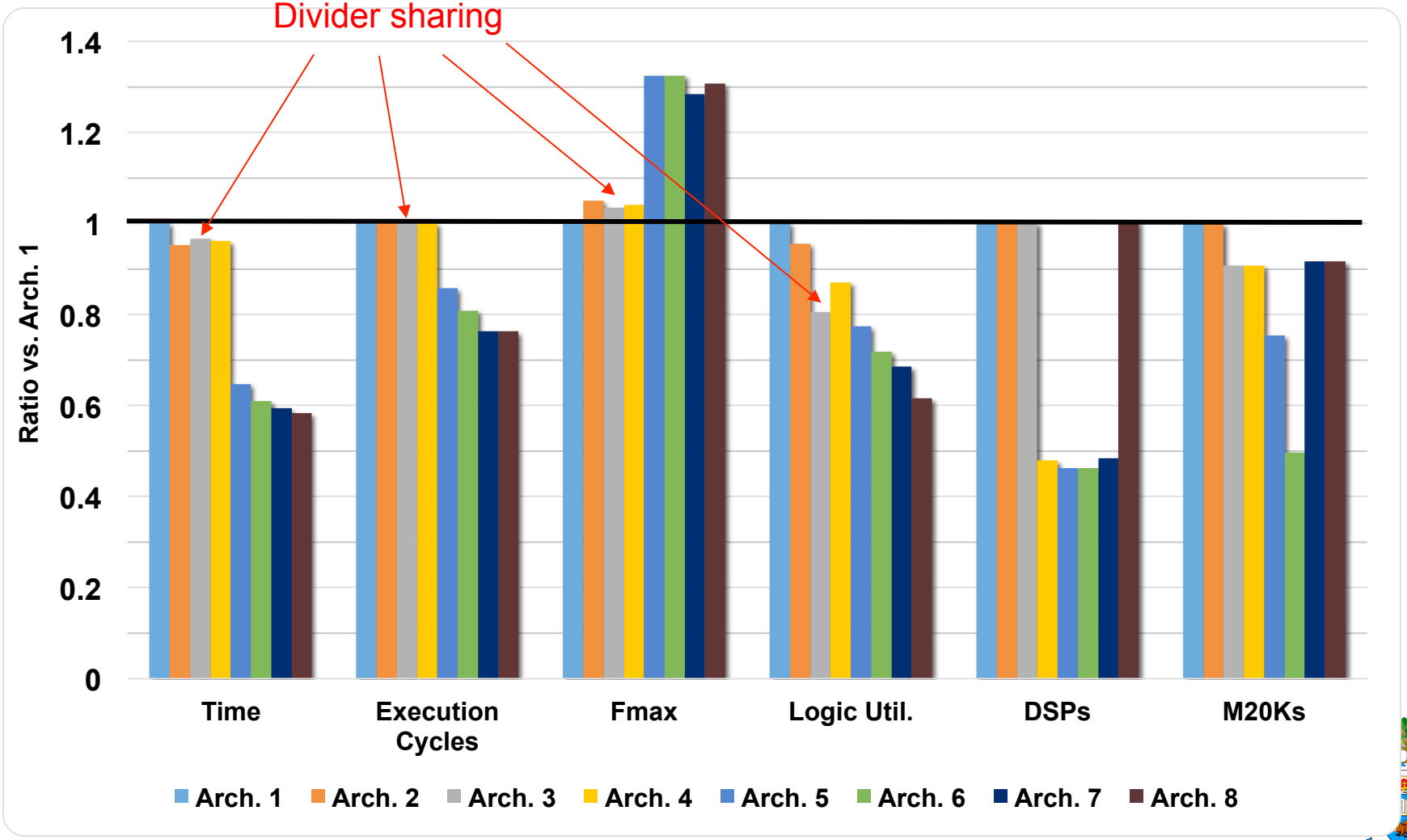
Benchmark	Description
Alphablend	Alphabends two images
Barrier	An accumulation benchmark which uses a barrier
Box Filter	A convolution filter commonly used in image processing
Dot Product	Dot product of two arrays
Histogram	Accumulates integers into 5 equally-sized bins
Matrix Multiply	matrix multiplication of two arrays
Mutex	An accumulation benchmark which uses a lock
Vector Add	Performs vector addition of two arrays

- Synthesized for Altera Stratix V FPGA with Quartus 15.0

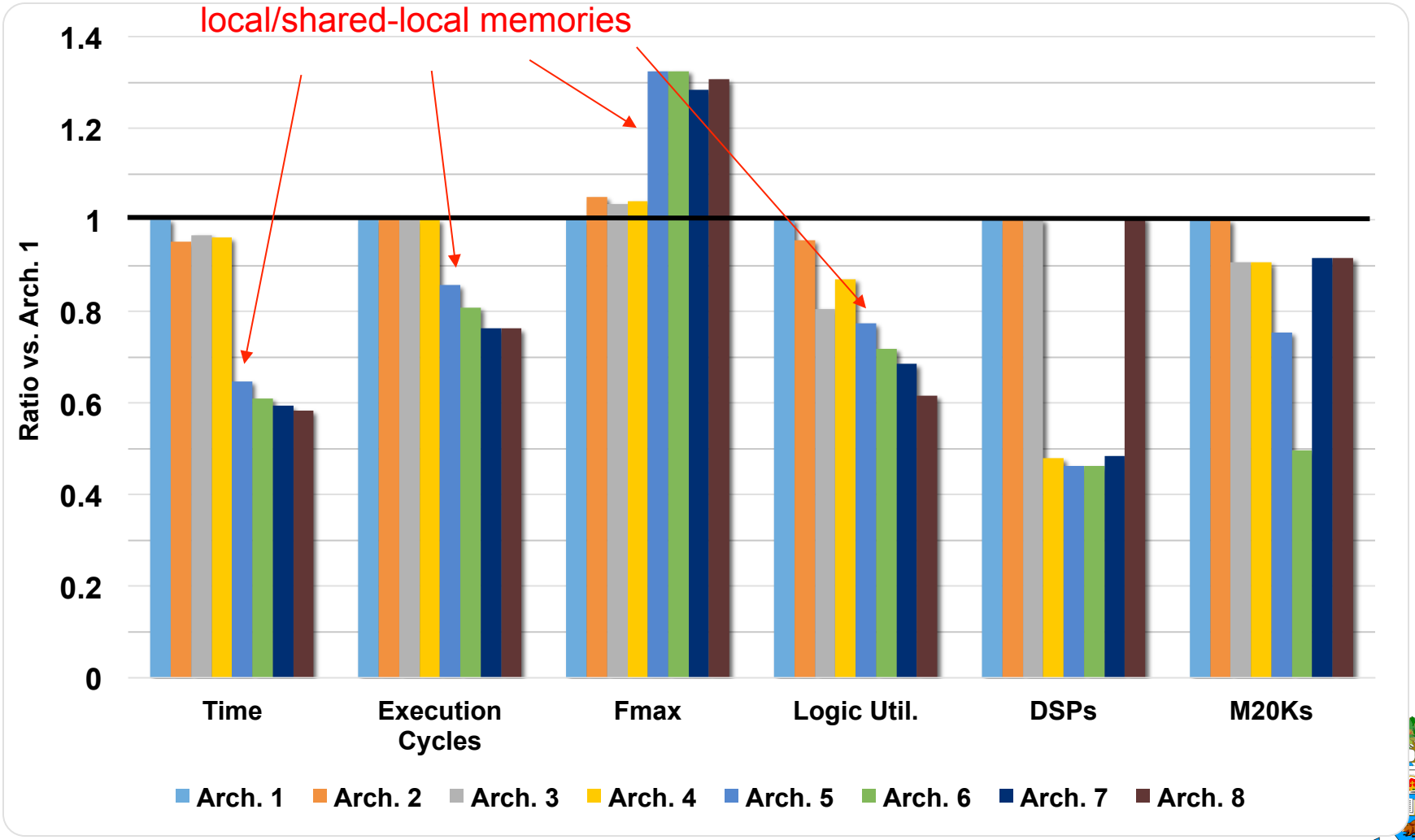
Geomean Performance/Area Results



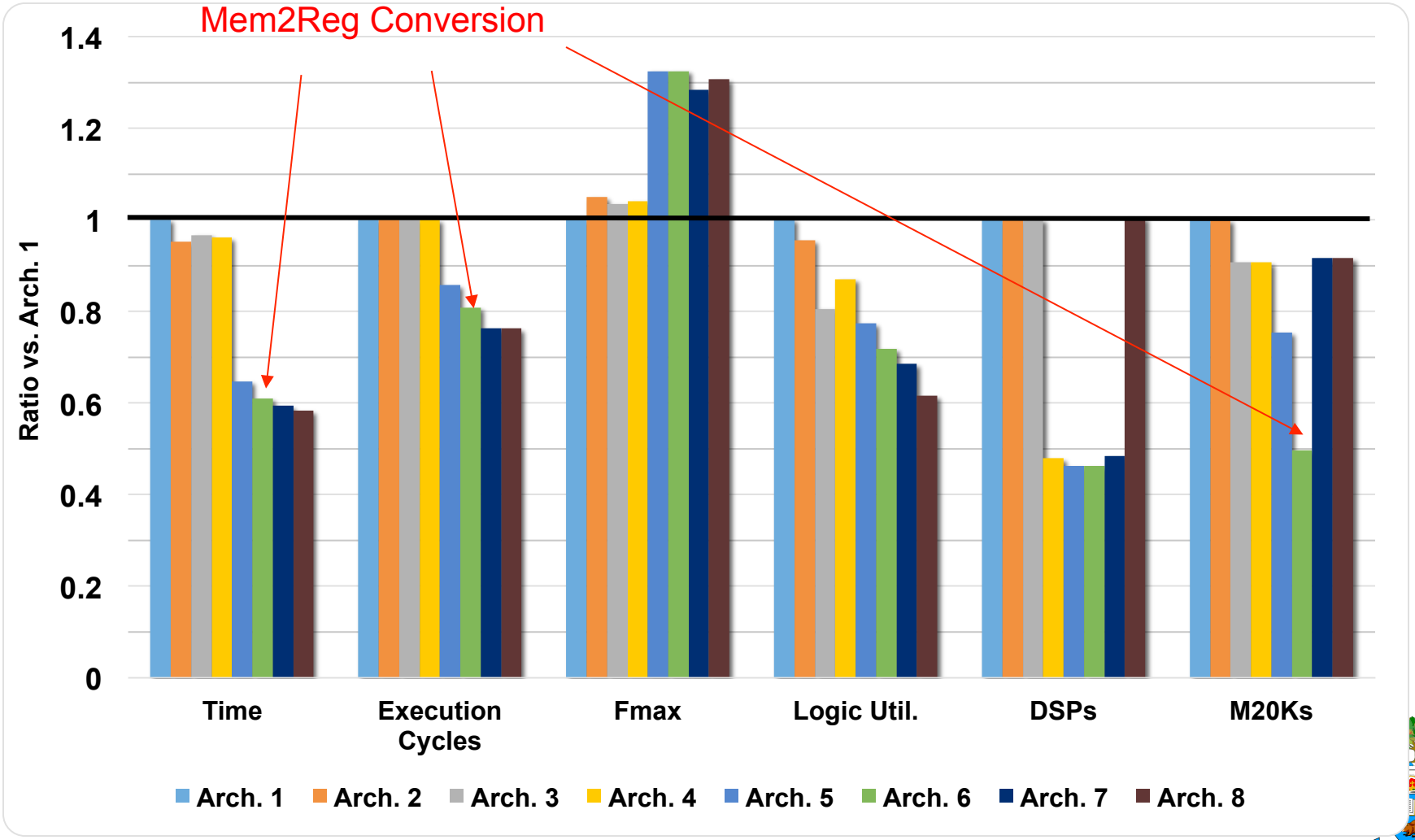
Geomean Performance/Area Results



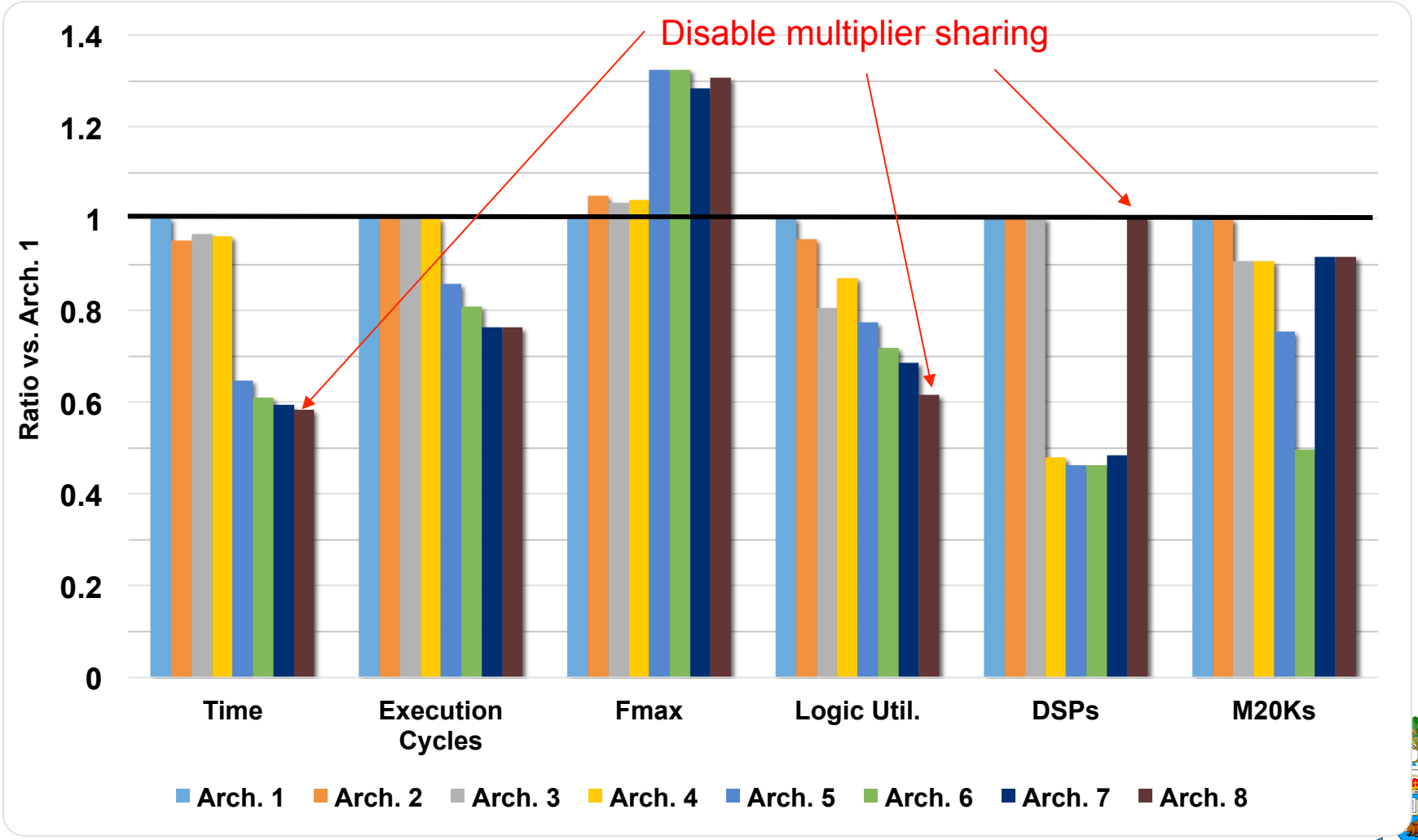
Geomean Performance/Area Results



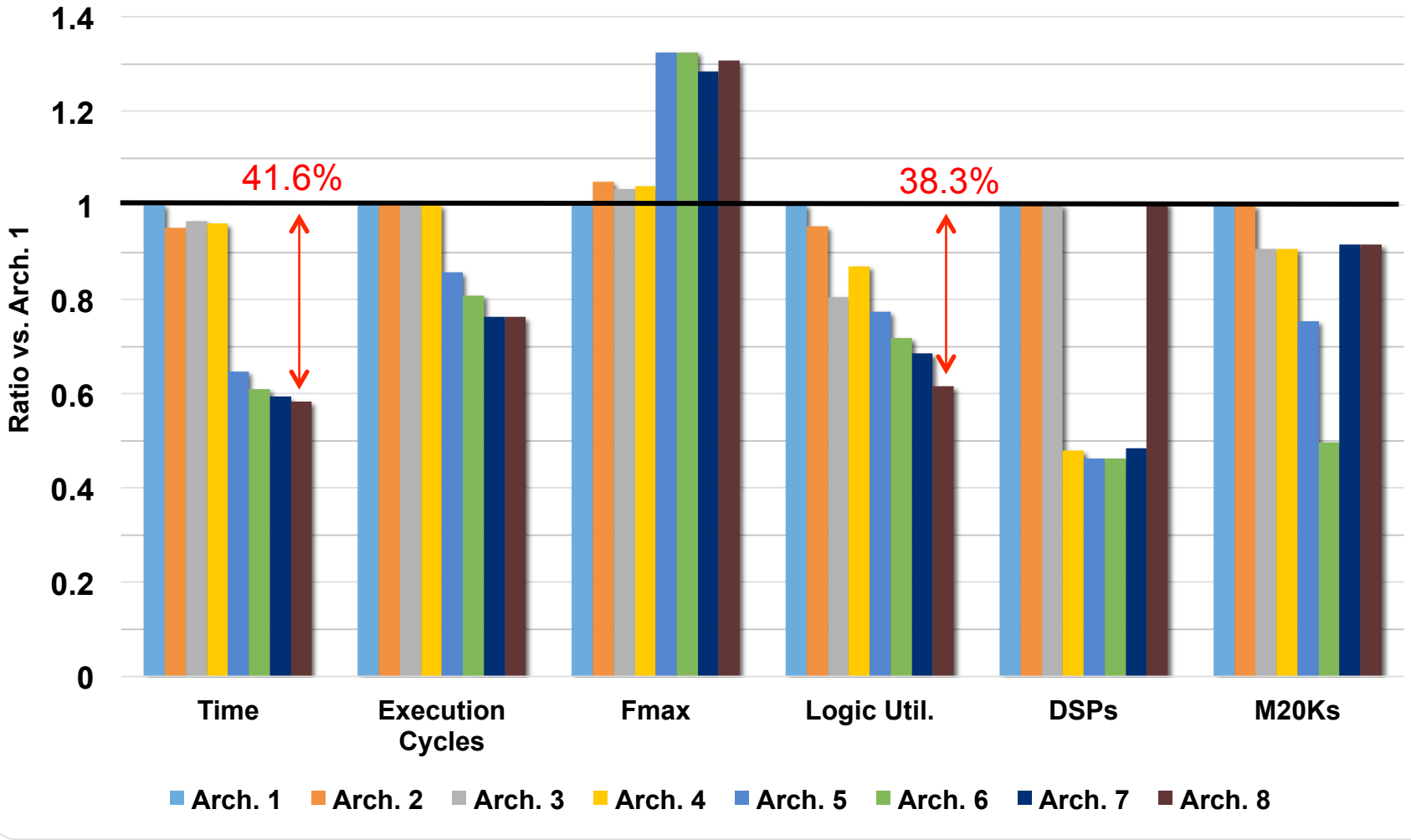
Geomean Performance/Area Results



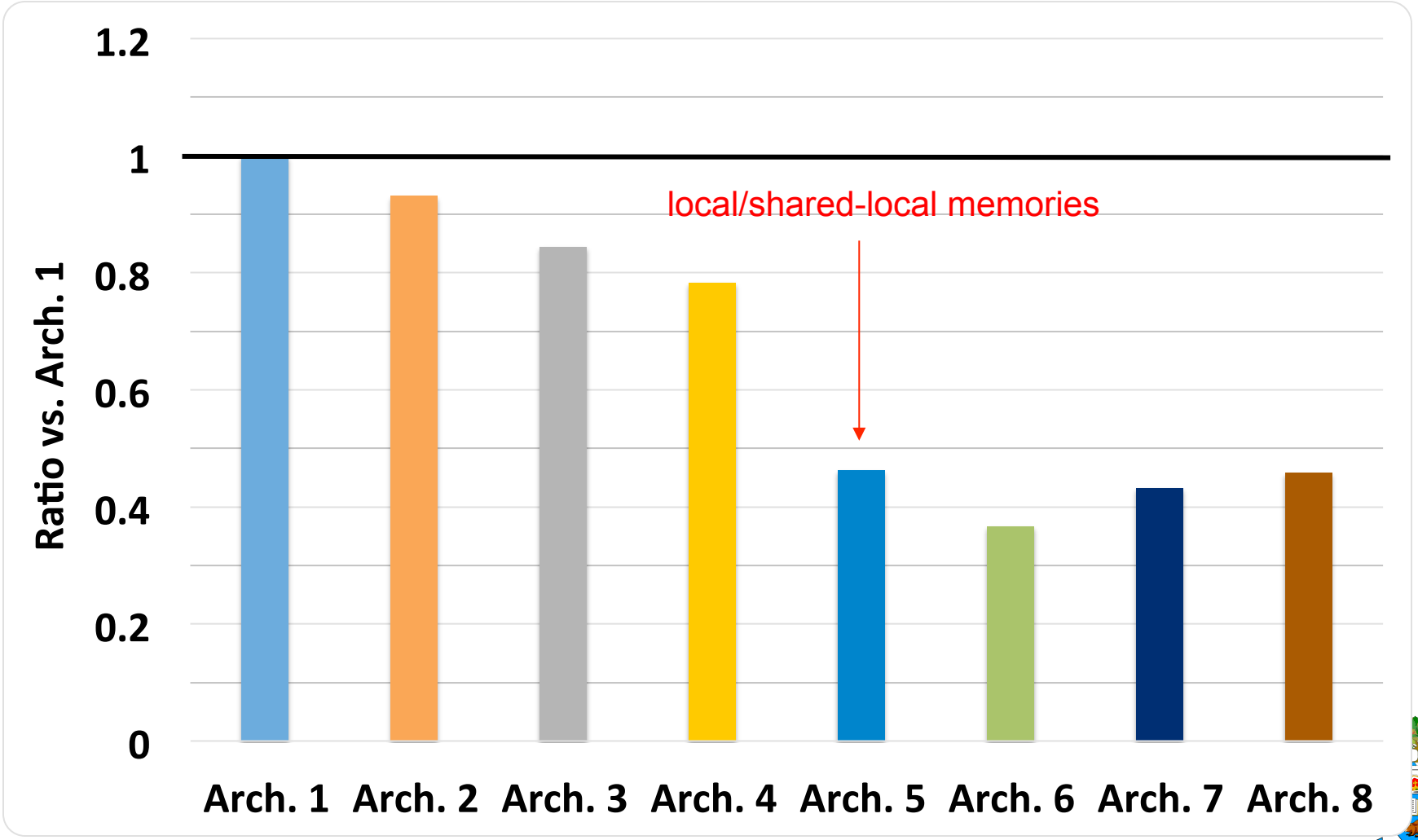
Geomean Performance/Area Results



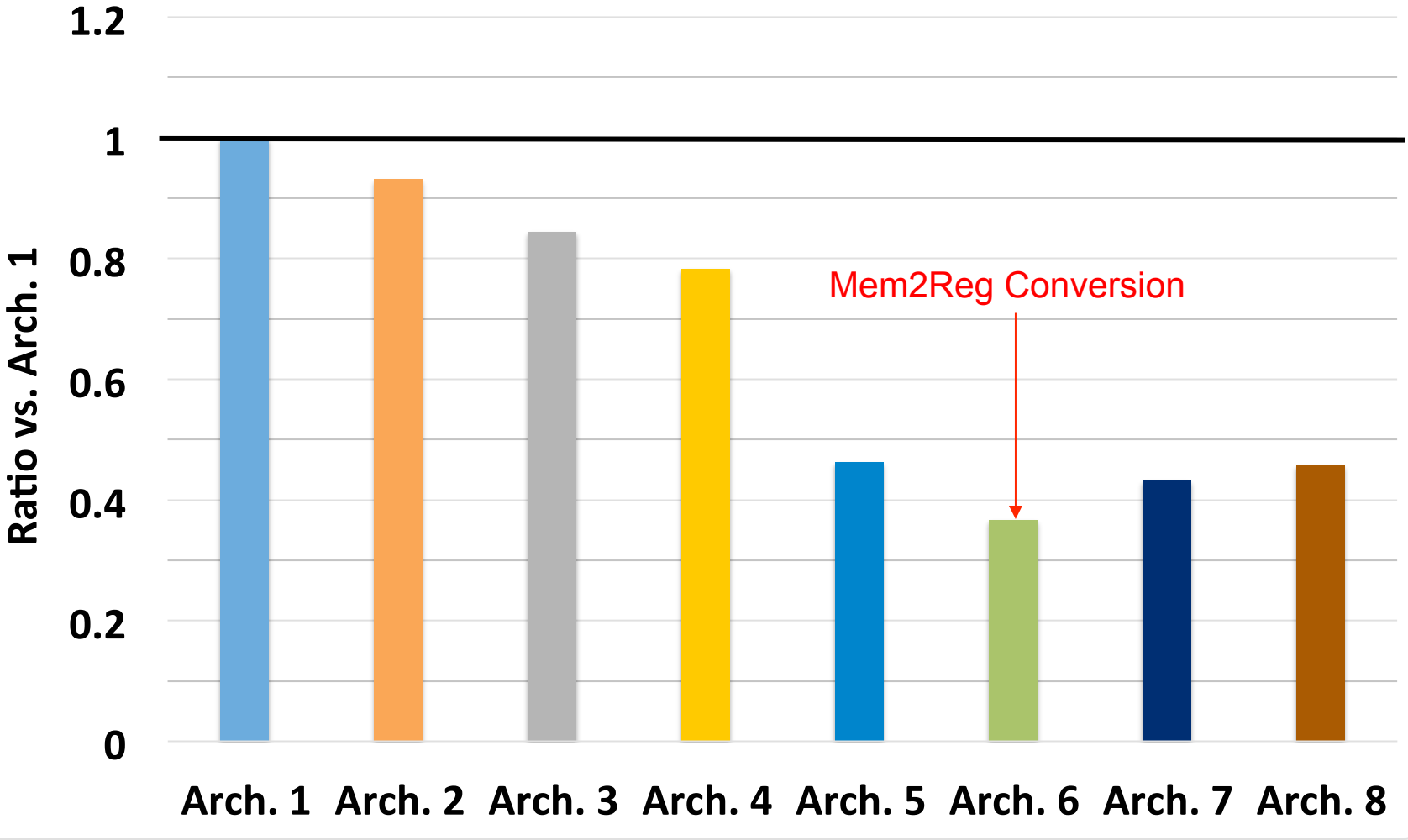
Geomean Performance/Area Results



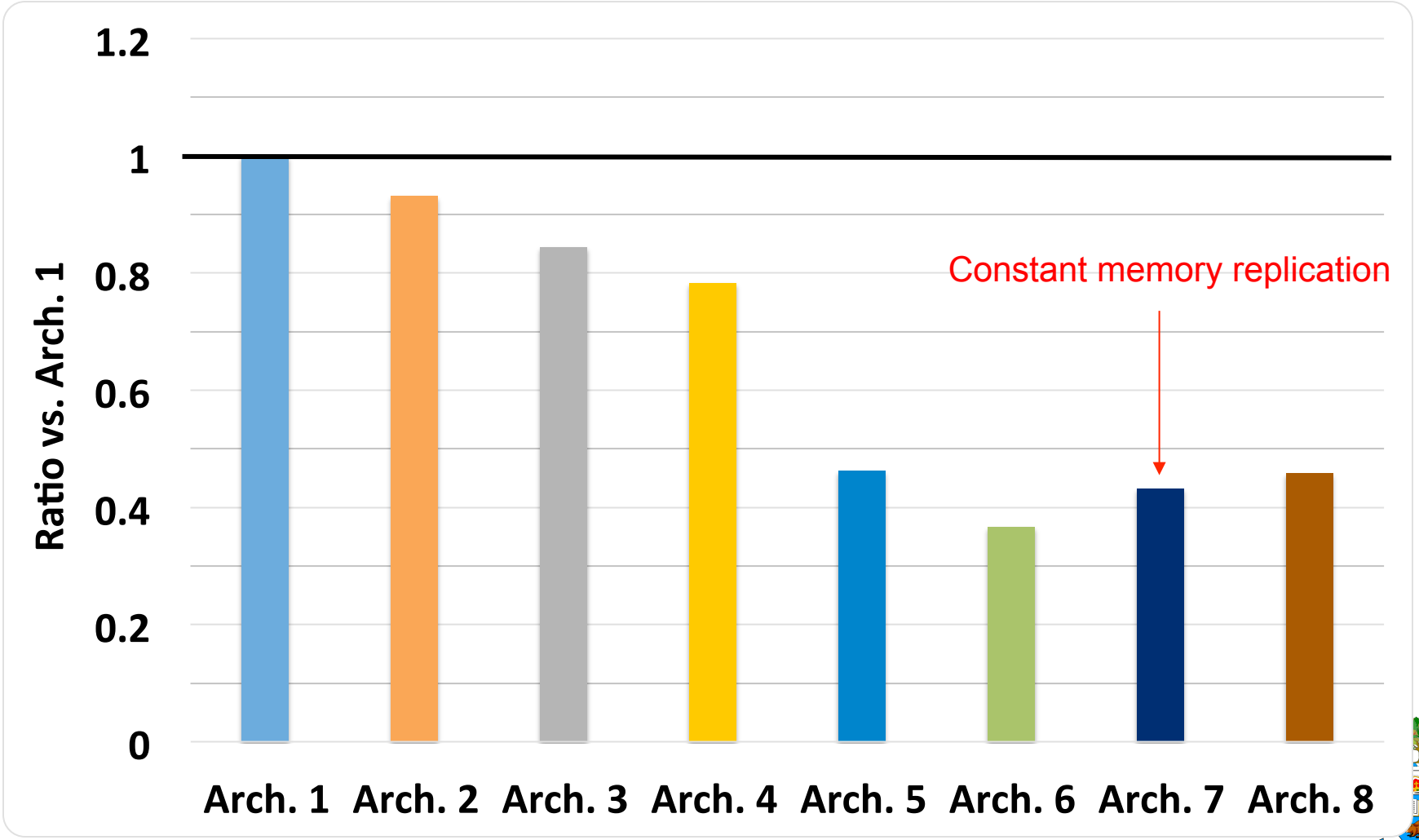
Geomean Area-delay Product



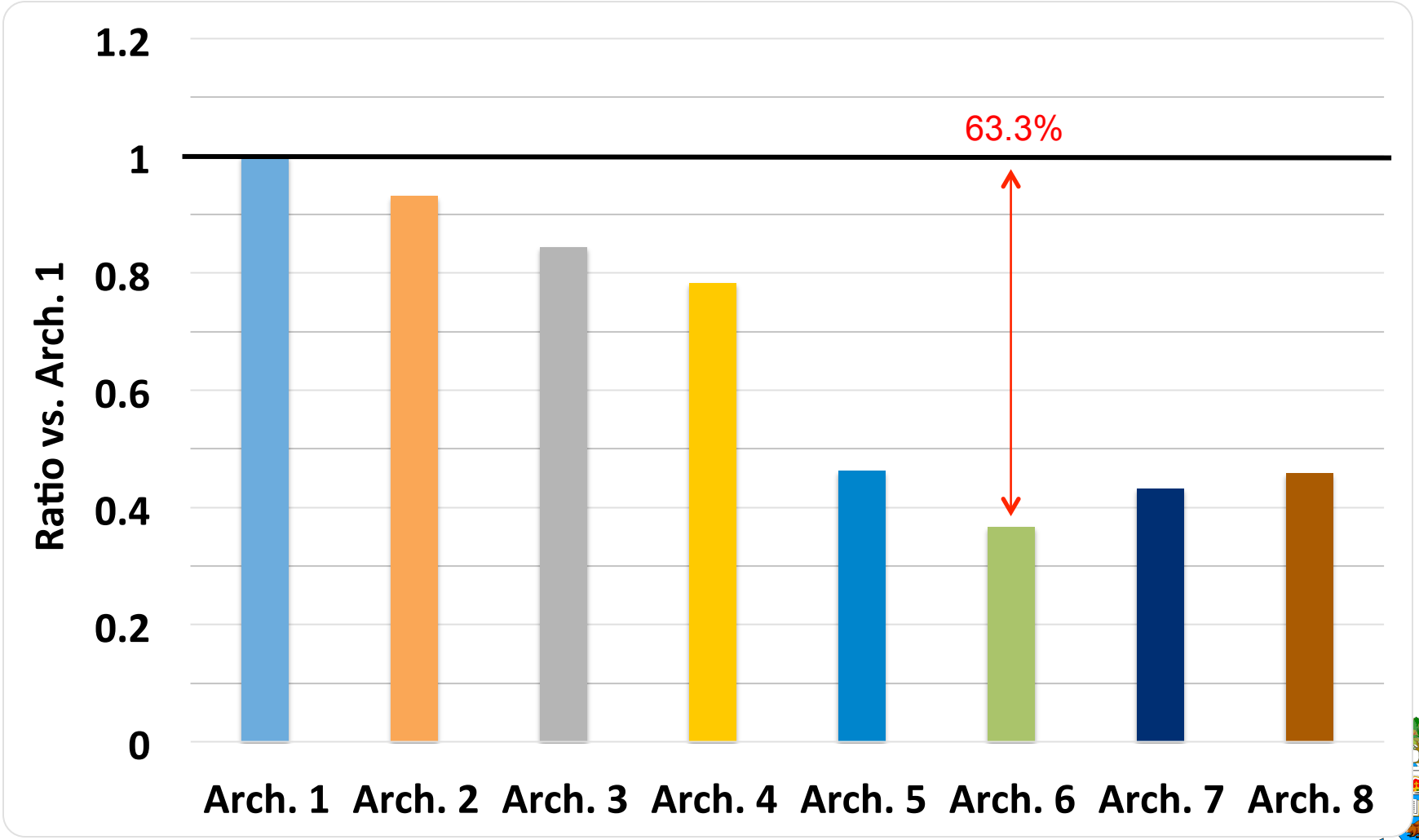
Geomean Area-delay Product



Geomean Area-delay Product



Geomean Area-delay Product



Summary

- Analyzed two different circuit topologies: Nested vs. Flat
- Presented a system generator which can:
  - Automatically share/replicate functions, functional units, memories
  - Inserts deadlock-prevention modules for parallel operating HW
- Investigated three different memory architectures
  - Global, local, and shared-local memories
  - Constant memory replication: reduces memory contention
  - Memory-to-register conversion: decreases memory usage and latency

Summary

- Local/shared-local memories significantly improved performance and area
- Constant memory replication reduces memory contention, but degrades area-delay product
- Sharing functional units across threads had little impact on performance, while producing considerable area savings

- Best results:
  - **Performance** (**41.6%** improvement): Flat with local/shared-local, divider sharing, mem2reg conversion, and constant memory replication (**Arch. 8**)
  - **Area-delay product** (**63.3%** improvement): Flat with local/shared-local, divider/multiplier sharing, and mem2reg conversion (**Arch. 6**)

Questions?