

C-slow Retiming of a Microprocessor Core

Yury Markovskiy
Yatish Patel

with input from
Nick Weaver

CS252
Semester Project

Outline

- Motivation
 - Alternatives
- C-slow + Retiming Transformations
 - An automatic mechanism to increase the clock rate
- Semantics of C-slowng a microprocessor design
 - Becomes a multithreaded machine
- Results of C-slowng LEON-1
 - A synthesized SPARC core

Motivation

- How to increase Instruction Throughput in a processor?
 - Tomasulo? Expensive and complicated!!!
 - VLIW? Also expensive and underutilizes hardware
 - Hyper-pipelining? Complex forwarding and hazard detection.
 - Replication
- Multi-threading (*independent* threads)
 - Simultaneous MT → Modern high-end processors (P4 Xeon)
 - Fine-grained MT → HEP, Tera
 - **C-slow + Retime** → Embedded, low cost systems
 - Ideally, straight forward to implement

Alternatives (1) – Replication

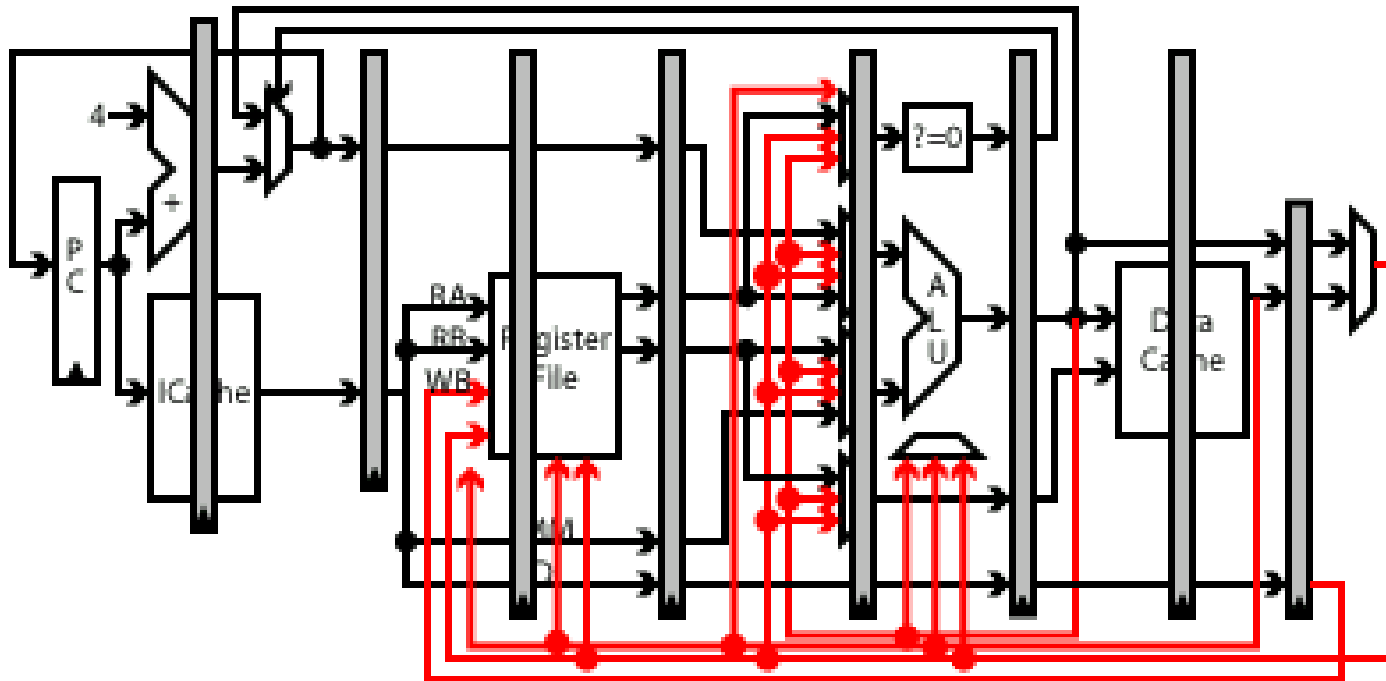
- Multiple cores on a chip → SMP
- Area = (# threads) * (core size)
- Requires synchronization/arbitration between cores
- Significant increase in cost

Alternatives (2) - Hyper-pipelining

- Expand bypassing and hazard detection
 - Increases complexity and area
 - More stages to forward from/to
 - Bypassing and control logic are a big portion of the area
 - Significant changes to the design
- Difficult to automate (vs. retiming)
- Potentially better single-thread performance
 - Easy to schedule a single process with *enough ILP*

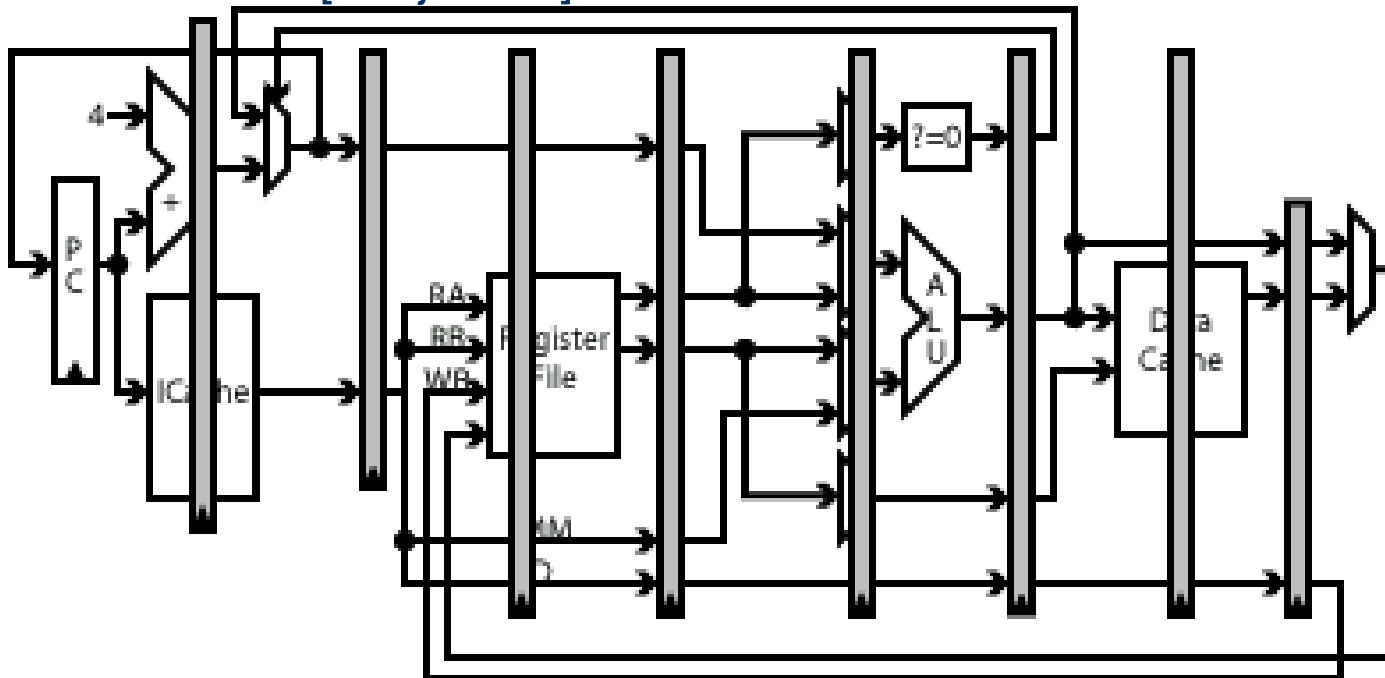
Hyper-pipelining Diagram

- Hyper-pipelining
 - Complex bypass logic and control
 - High area and performance cost (wire dominated delays)



Hyper-pipelining - No Bypass

- Multiple threads \rightarrow No bypass logic
 - Performance degradation when number of threads is low
 - Removal of bypass logic results in 20% to 80% performance drop on SPECint92 [Ahuja et al]

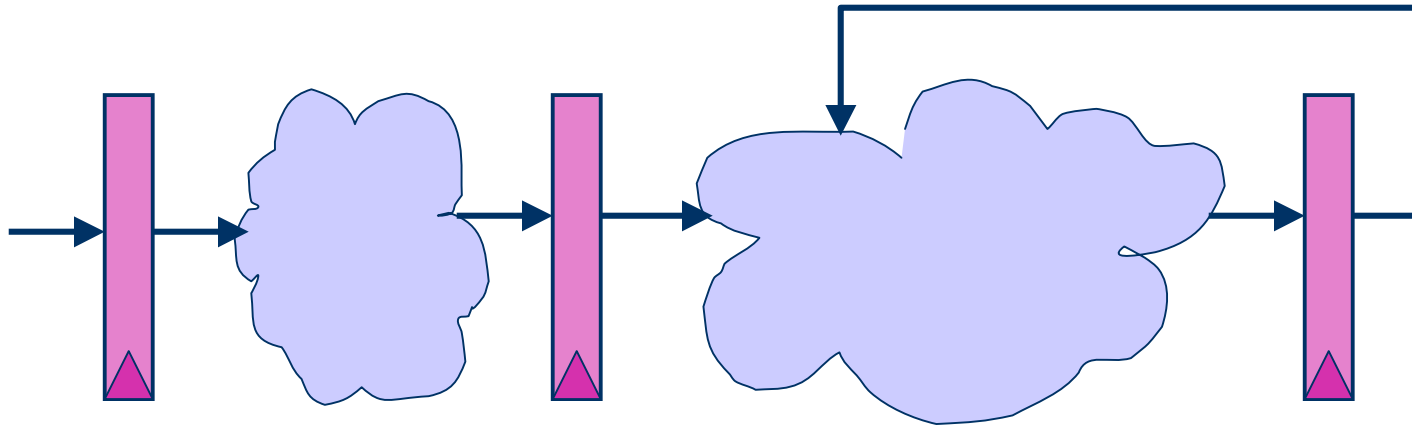


C-Slow + Retime

- Existing processor's IPC remains the same
 - *No change* to bypass/forwarding logic
- Transformation can be performed automatically
- Minimal increase in area
 - Avoids unnecessary replication of resources
- Straightforward SMP/multi-threading semantics

C-Slow transformation (1)

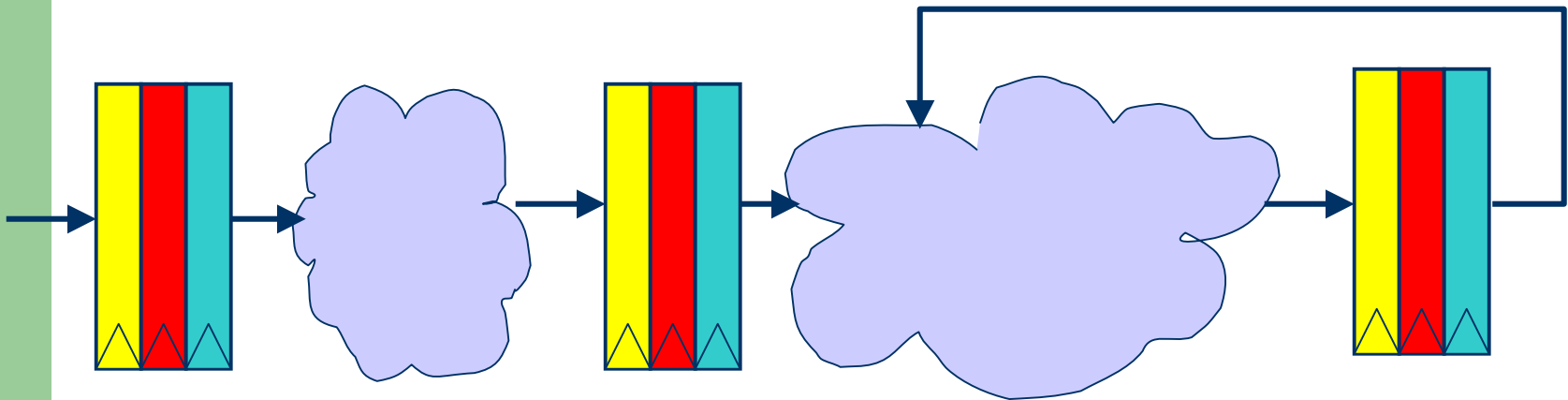
- Applicable to designs limited in throughput by feedback cycles
 - Forwarding, Hazard detection, and Control logic



Leiserson, Saxe “Optimizing Synchronous Circuitry by Retiming” 1981

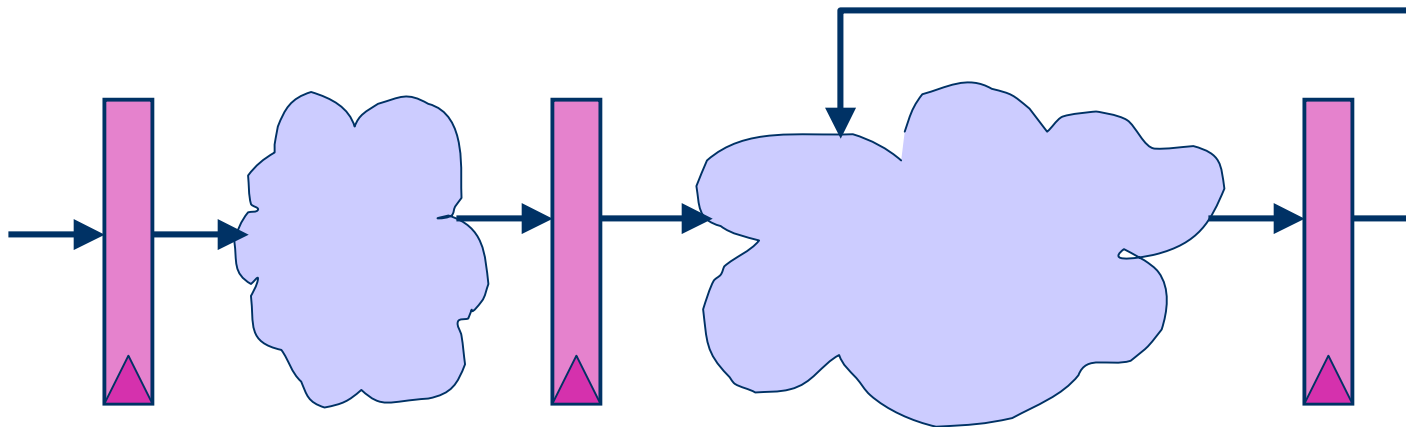
C-Slow transformation (2)

- Replace every register with C registers
 - Interleave C independent data streams
 - *i.e.* threads



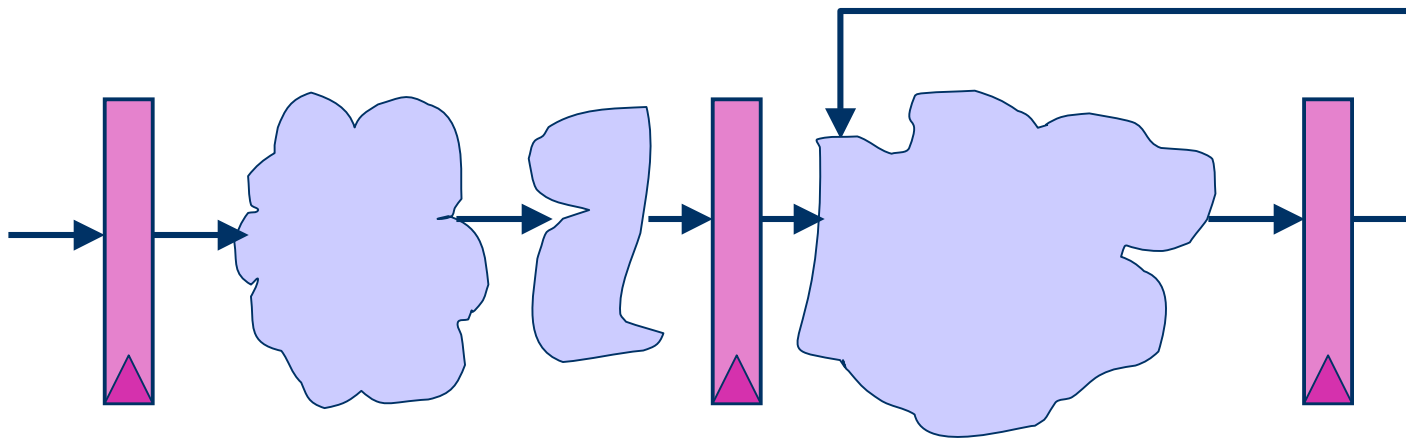
Retiming

- An automatic process of moving registers to balance delays in the critical path
 - Supported (poorly) by many HDL synthesis tools



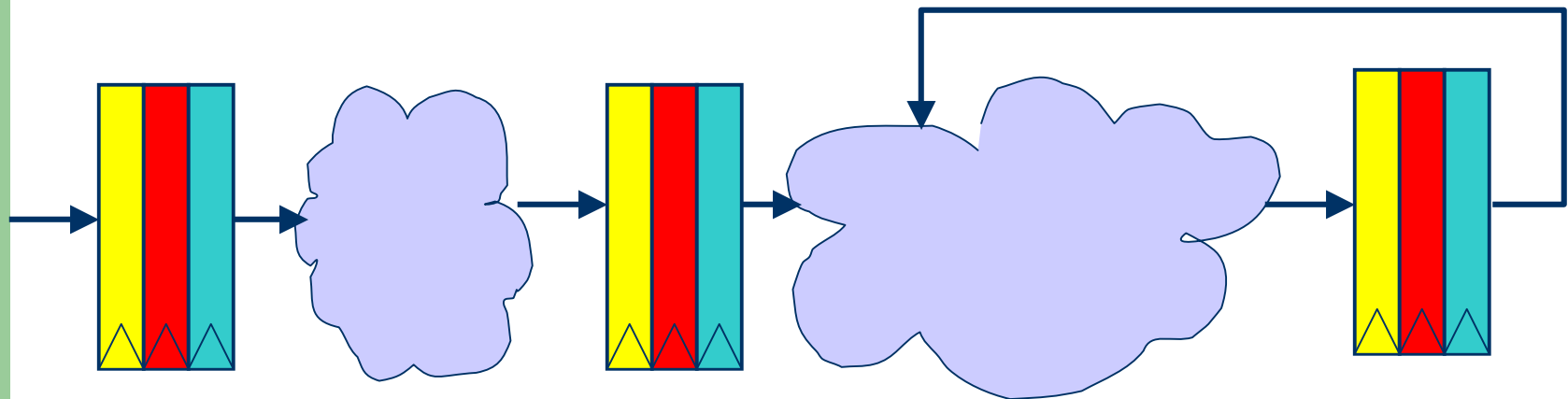
Retiming

- An automatic process of moving registers to balance delays in the critical path
 - Supported (poorly) by many HDL synthesis tools



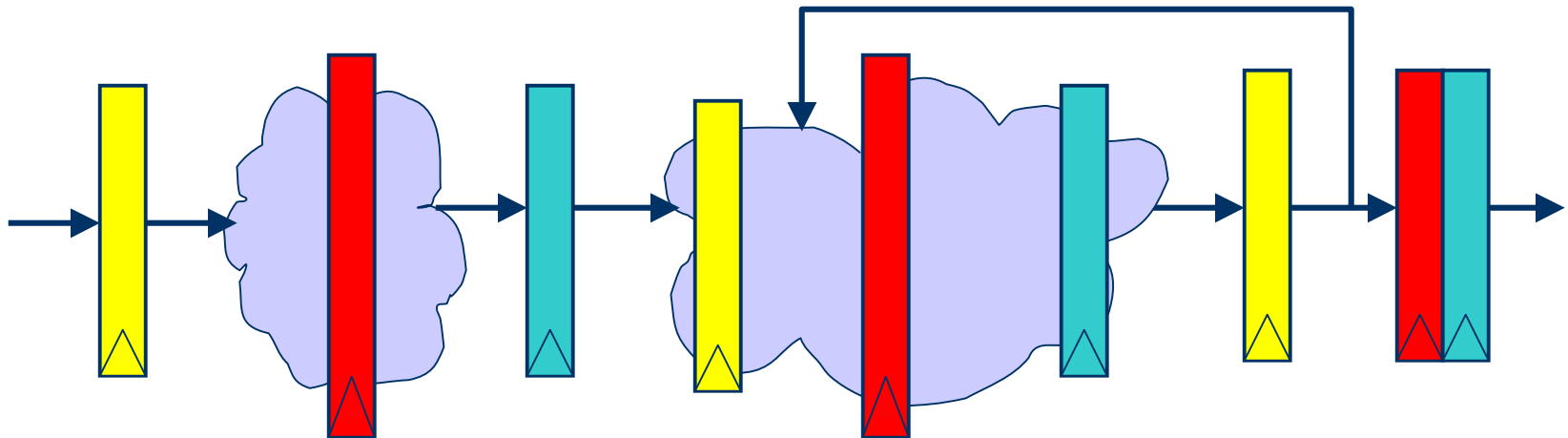
C-Slow + Retiming

- Increases clock rate and throughput on many designs
 - Works well when throughput is the primary goal
- No interaction between *independent* threads
 - Same control complexity as original design



C-Slow + Retiming

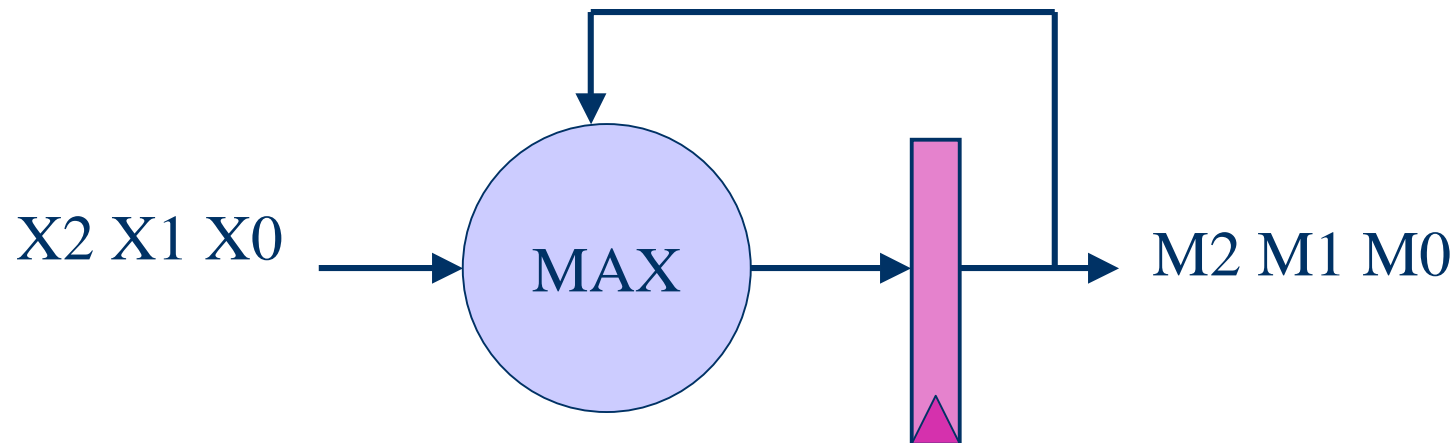
- Increases clock rate and throughput on many designs
 - Works well when throughput is the primary goal
- No interaction between *independent* threads
 - Same control complexity as original design



C-Slow + Retiming Example (1)

- Max Function

- $T_{\text{cycle}} = T_{\text{CL}} + T_{\text{su}} + T_{\text{cko}}$

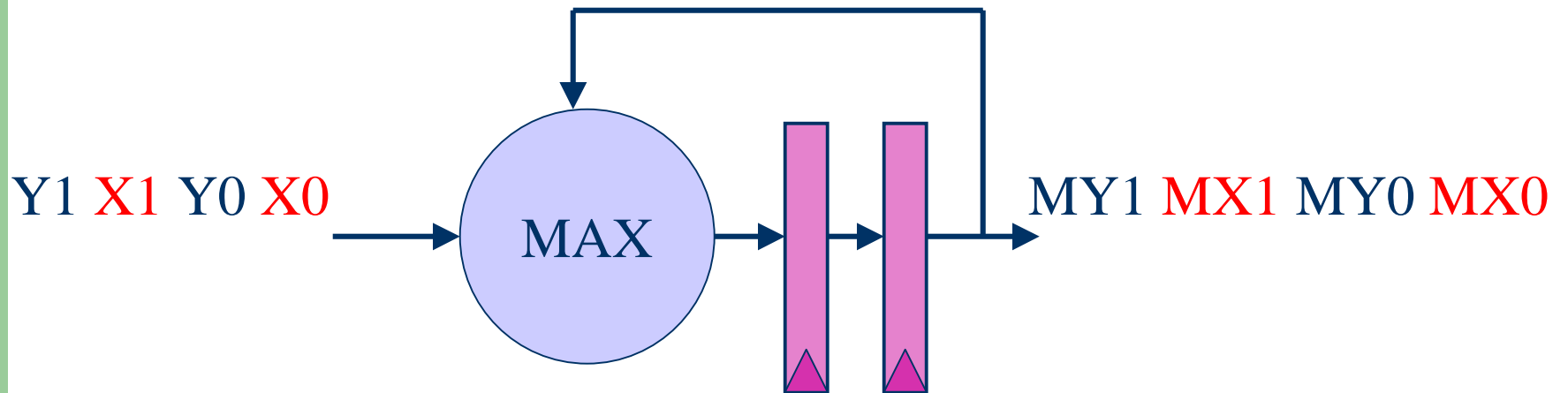


Example by André DeHon

C-Slow + Retiming Example (2)

- Max Function – 2-Slow

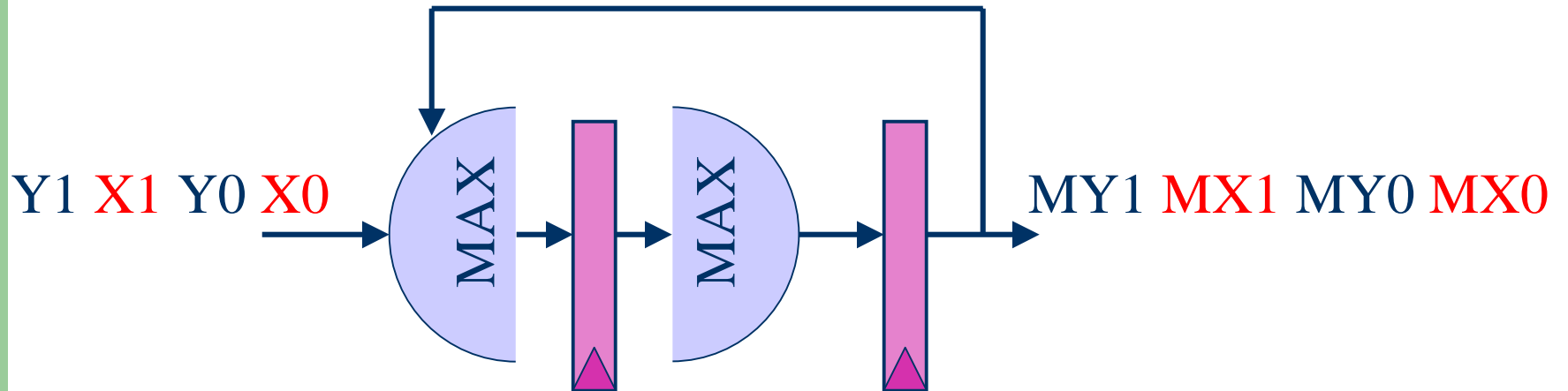
- $T_{\text{cycle}} = T_{\text{CL}} + T_{\text{su}} + T_{\text{cko}}$



C-Slow + Retiming Example (3)

- Max Function – 2-Slow

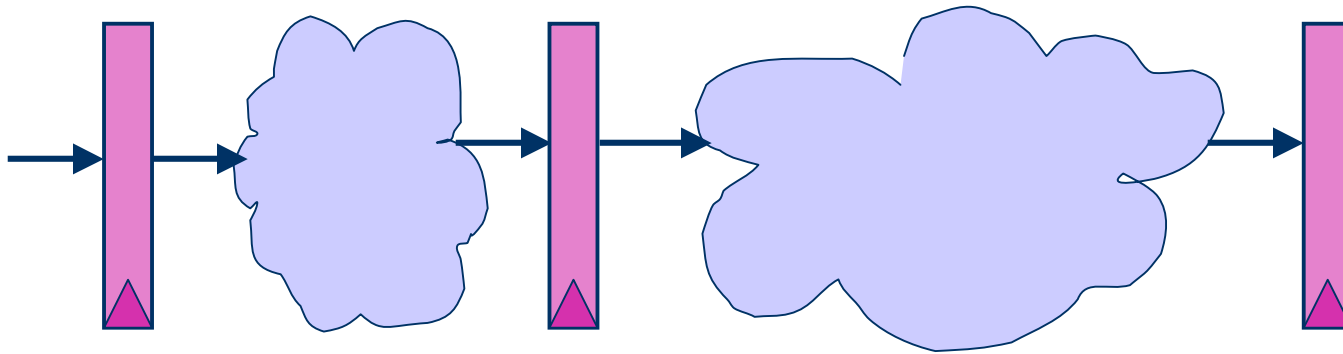
$$- T_{\text{cycle}} = T_{\text{CL}} / 2 + T_{\text{su}} + T_{\text{cko}}$$



- Ideal Improvement in throughput of $\sim 2x$

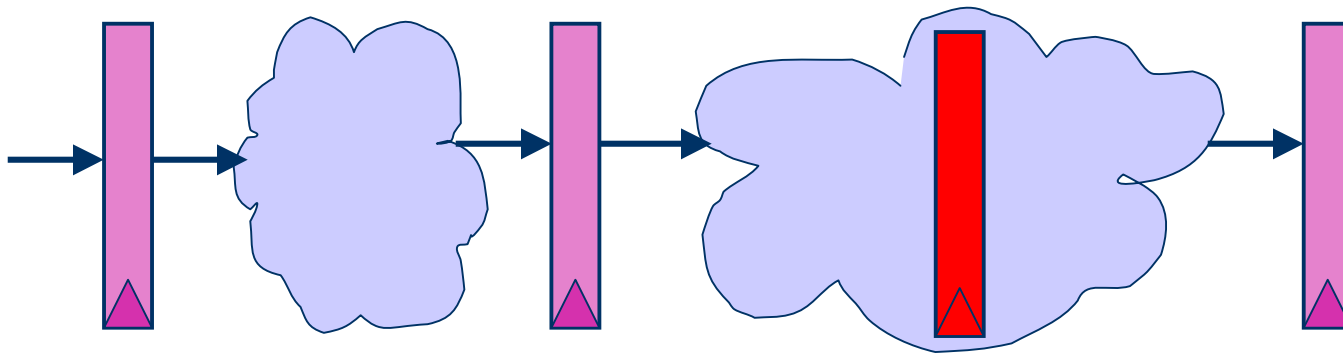
Limitation of retiming (1)

- Can only balance existing delays
 - Can't add additional pipeline stages to a design



Limitation of retiming (1)

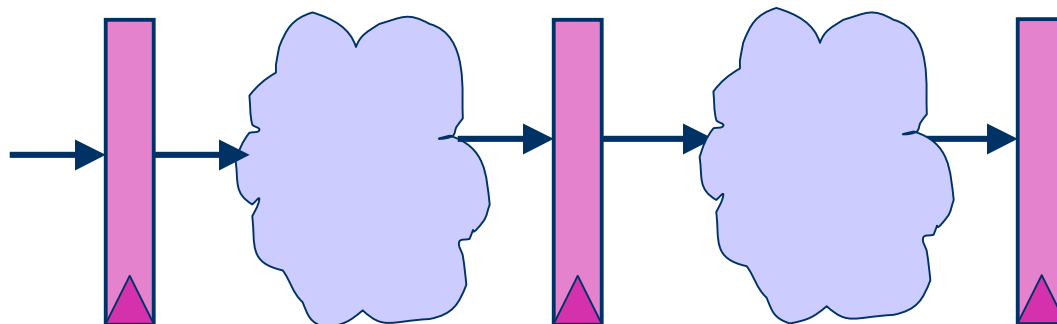
- Can only balance existing delays
 - Can't add additional pipeline stages to a design



- Pipeline latency must remain the same

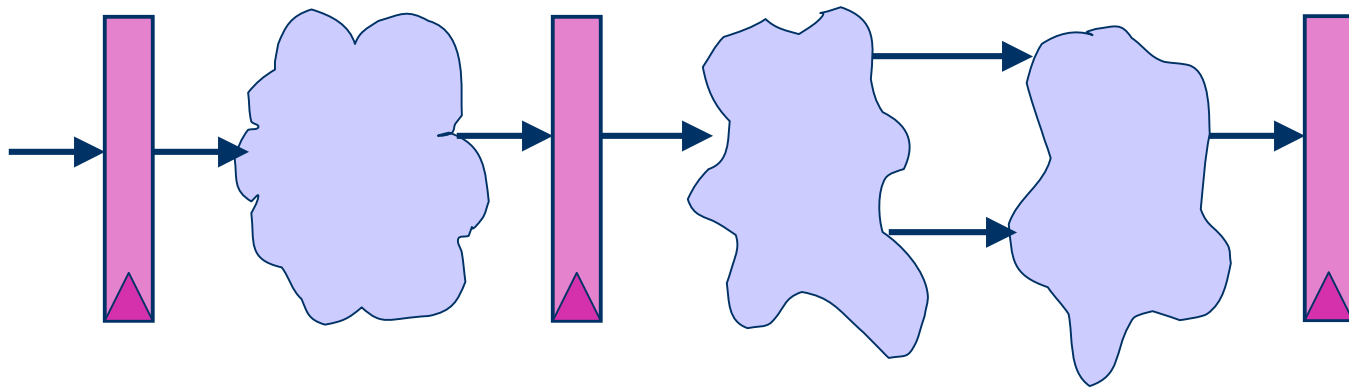
Limitation of retiming (2)

- Well constructed designs do not benefit from conventional retiming
 - Pipeline stage delays are already properly balanced



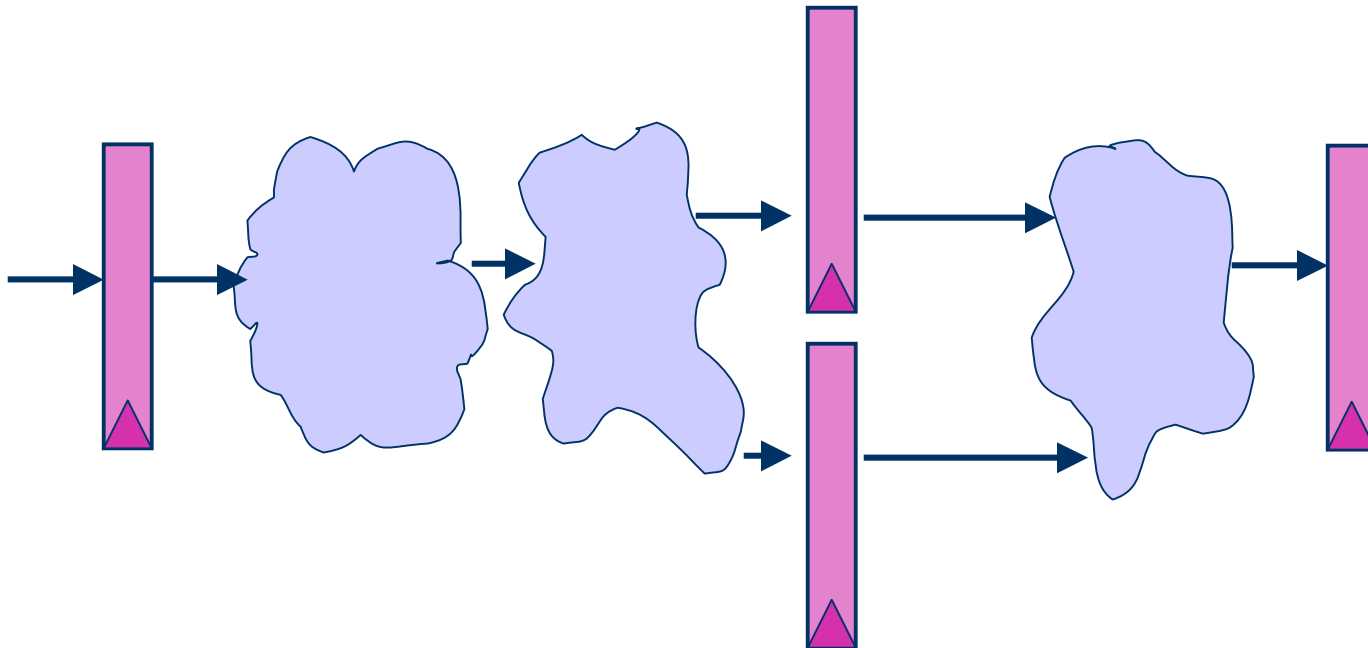
Limitation of retiming (3)

- May increase the number of registers required for a design
 - When a register is pushed through net “fan out”



Limitation of retiming (3)

- May increase the number of registers required for a design
 - When a register is pushed through net “fan out”



Limitations of C-slow (1)

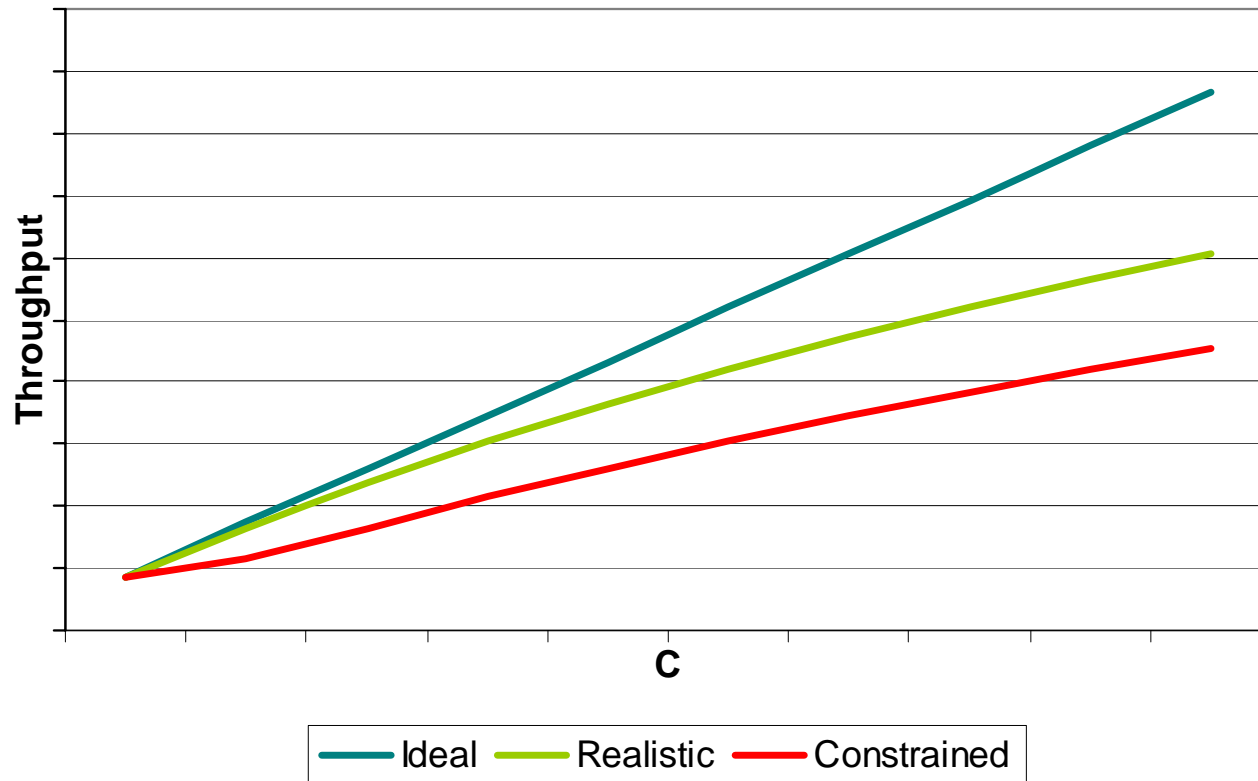
- Requires application to support multiple data streams
 - e.g. encryption, multimedia, multithreading
- Requires significantly more registers
 - *However*, matches FPGA architectures well
 - Equal number of flip-flops as lookup tables (logic)
 - Enabling technology for *fixed frequency* FPGAs (e.g. HSRA)
- Requires more power
 - More registers, eliminates data correlation

Limitations of C-slow + Retiming (2)

- Given a critical path and clock period of
 - $T_{\text{cycle}} = T_{\text{CL}} + T_{\text{su}} + T_{\text{cko}} + T_{\text{skew}}$
- A C-slow design would have an ideal clock period of
 - $T_{\text{cycle}}(C) = T_{\text{CL}}/C + T_{\text{su}} + T_{\text{cko}} + T_{\text{skew}}$
- And a best case single thread latency of
 - $T_{\text{CL}} + C(T_{\text{su}} + T_{\text{cko}} + T_{\text{skew}})$
 - because of the increased number of registers

Limitations of C-slow + Retiming (3)

MultiThread Throughput



C-slowng a microprocessor

- Why?
 - Demonstrate that even a complicated design can be successfully C-slowng
 - Many microprocessor workloads already support multiple threads
- Semantics work!
 - C-slowng processor behaves like a multithreaded machine or SMP
- Target: low cost embedded microprocessor

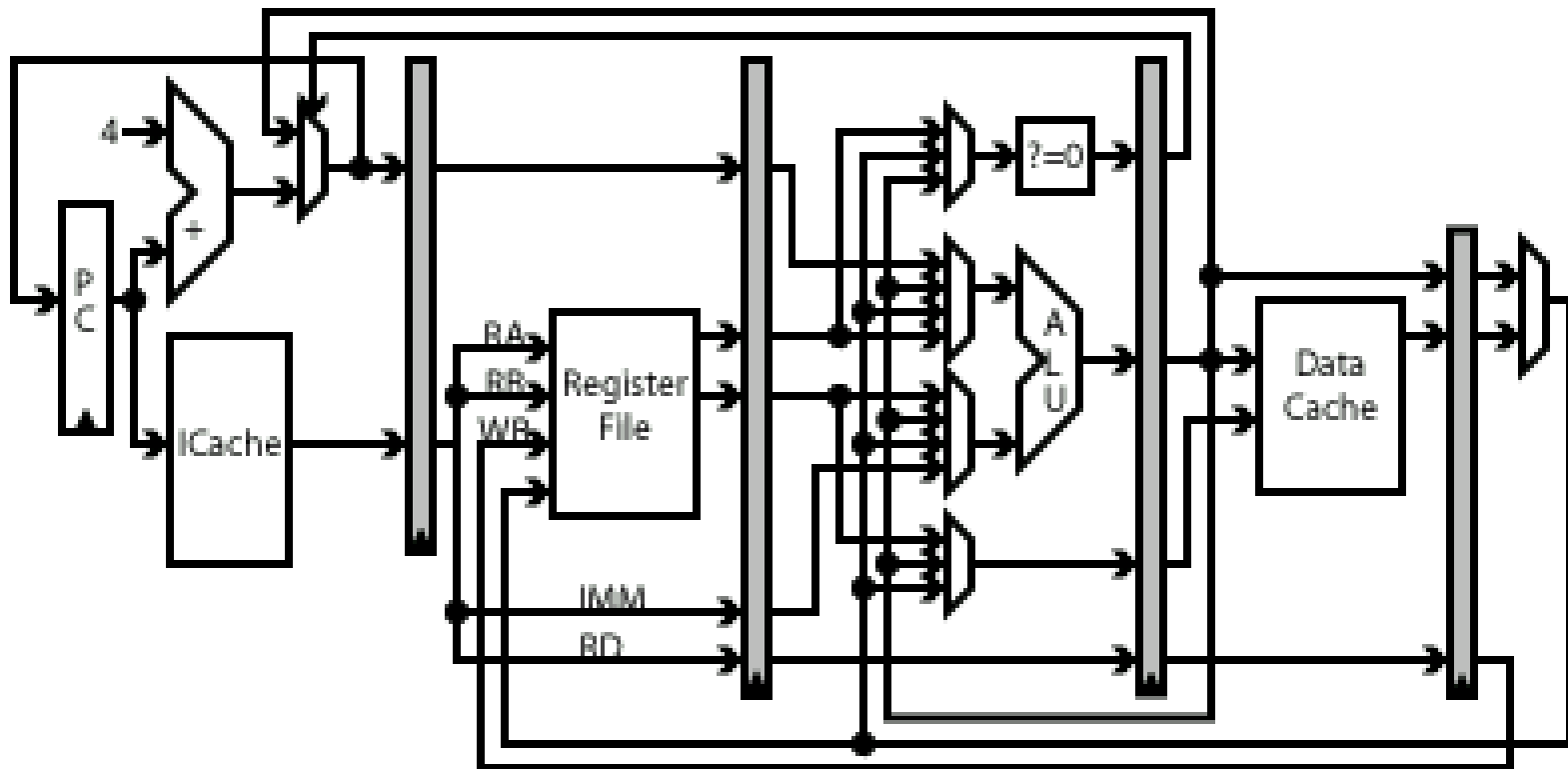
How to C-slow a processor?

- C-slow transformation: storage elements $\times C$
 - Pipeline stage registers
 - Status registers
 - Register file
 - Each thread has its own register set
 - Cache
 - Increase associativity or size
 - Tag memory transactions by a thread ID
 - Disambiguate the separate threads of executions
- Note:
 - **Multi-thread IPC is the same as the original IPC**

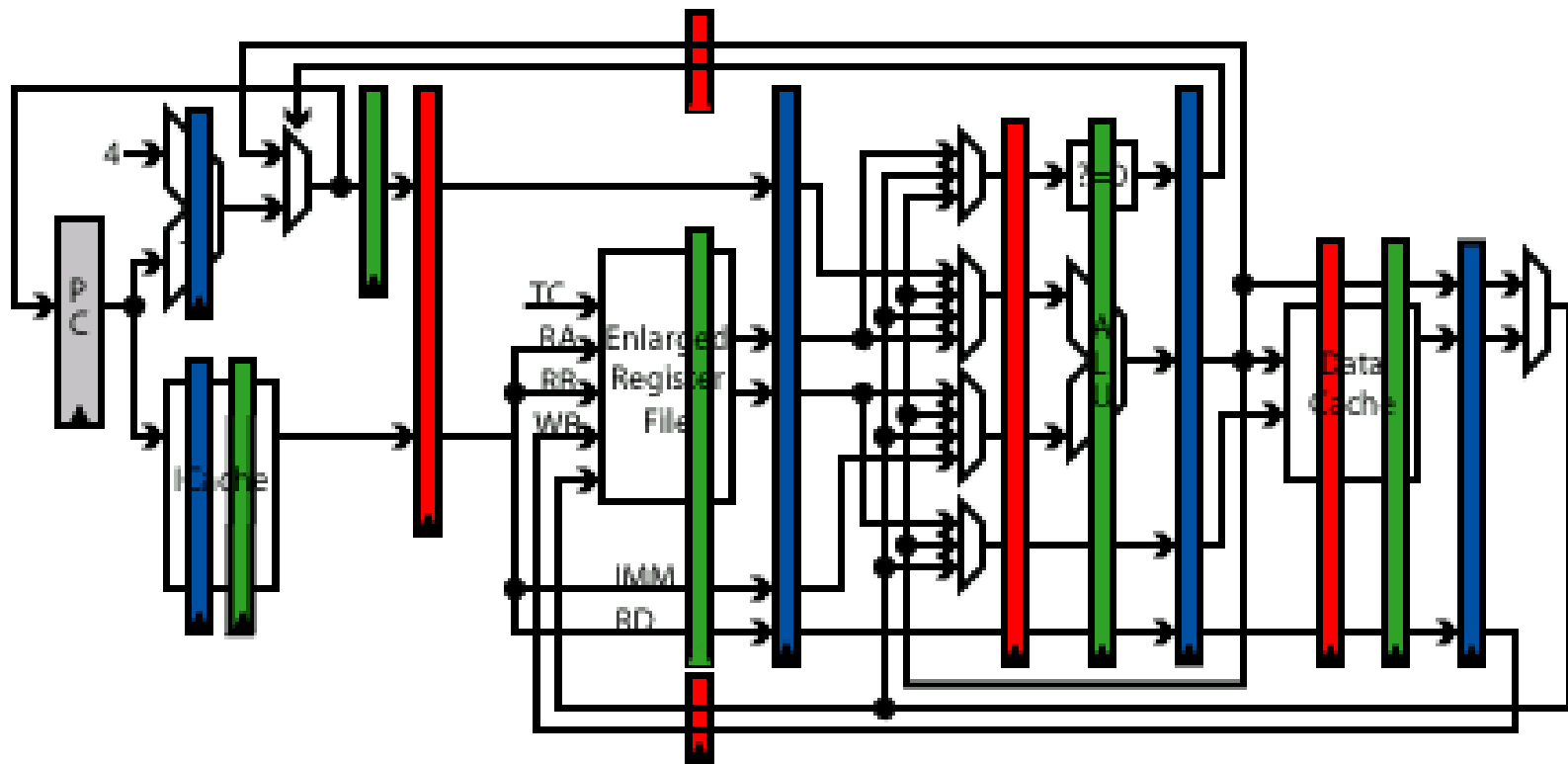
Continue with retiming

- Retime:
 - Balance pipeline delays
 - Increase maximum frequency
- **Increase in throughput**
 - Multi-thread IPC is the same as the original IPC
 - Multi-thread $IPT(C) = IPC / T_{\text{cycle}}(C)$
 - Single thread $IPT(C) = IPC / (C * T_{\text{cycle}}(C))$

Processor C-slow + Retime



Processor C-slow + Retime



→ Simple Multithreading

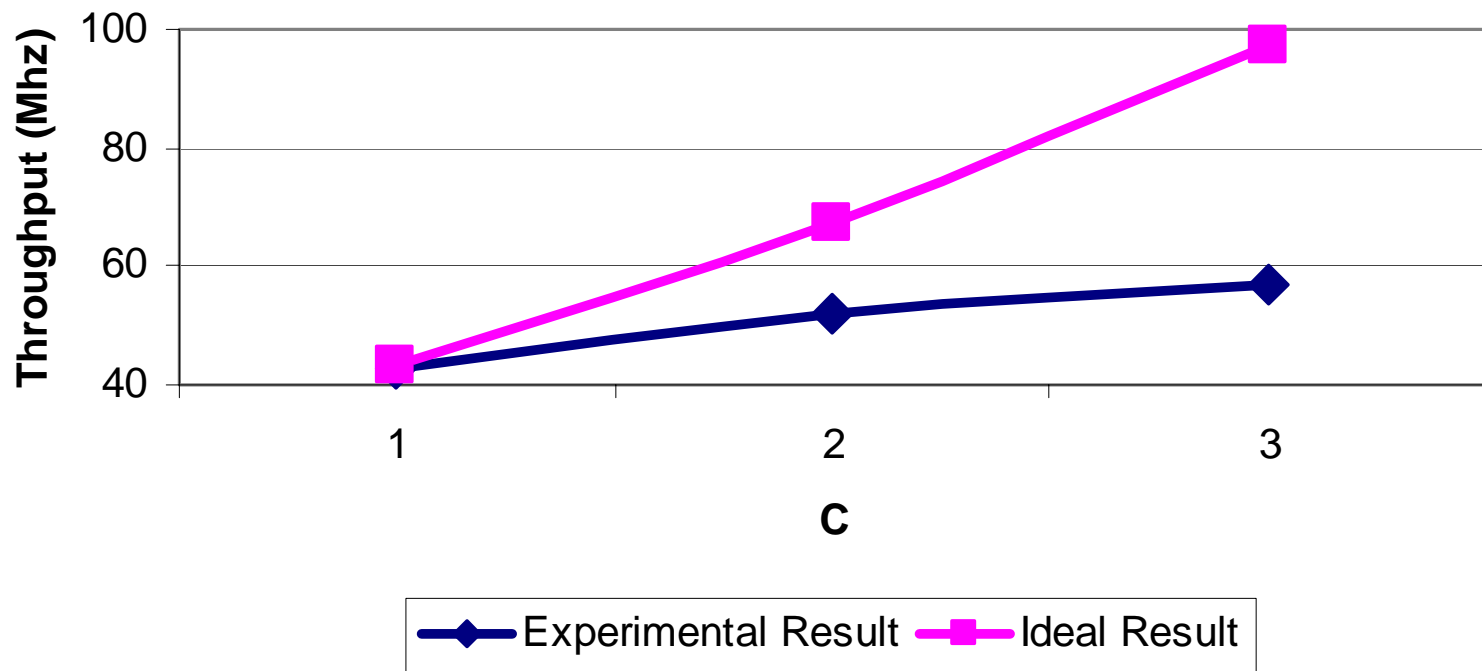
- Each thread has its own register file
- Each thread also has its own interrupt vector and status registers
 - Alternative: use a single thread to handle IO tasks
- All threads share caches
 - Problem: thrashing (destructive interference)
 - Limit thrashing by reserving associativity sets for specific threads when redesigning the cache
 - Benefit: synergy (constructive)
- Must tag memory transactions with Thread IDs.

Our experiment

- LEON SPARC processor written in synthesizable VHDL
 - SPARC V8 compatible with 5 stage pipeline
 - Modified to add a customizable C factor of 1 to 3
- Synplify Pro HDL synthesis tool
 - Supports limited retiming on conventional VHDL/Verilog code
- Xilinx Virtex FPGA target - XCV800
 - Register rich FPGA family

Results(1) – Instruction Throughput

C-slow LEON Instruction Throughput



Results(2) – Instruction Throughput

- Multi-thread IPC = original IPC
- Compare clock frequencies
 - Directly proportional to the instruction throughput

C	Throughput (Mhz)			
	MultiThread		Single Thread	
	Absolute	Normalized	Absolute	Normalized
1	43	1.00	43	1.00
2	52.2	1.21	26.1	0.61
3	56.7	1.32	18.9	0.44

Results (3) – Area

- Data-path + Control:

C	LUTs (logic)			FF			Slices
	Absolute	Normalized	% of chip	Absolute	Normalized	% of chip	
1	1222	1.00	6	602	1.00	3	876
2	1522	1.25	8	1366	2.27	7	1266
3	1526	1.25	8	1926	3.20	10	1541

- Memory size (BlockRAM modules) increases by C
 - register file
 - caches

Limitations of Synplify's retiming

- Ideally: C-slow and retiming are automatic
- In reality: C-slow transformation is *not* supported
 - Synplify supports retiming
- Limitations:
 - Cannot retime across BlockRAMs
 - Used for caches and register files
 - Addressed by manually moving registers
 - Cannot retime across registers with a “clock enable”
 - Optimization routines take precedence
 - >2 registers in series automatically converted to shift registers preventing retiming
 - Recoded registers to trick Synplify into not optimizing

Future work

- Automated C-slow retiming tool
 - *Currently in progress* (difficult to interface with Xilinx tools)
 - Takes modules post synthesis and retimes them
 - Uses separate clock domains to determine which to C-slow and which to keep constant
 - Intelligent handling of shared memory elements