ECE 636

Reconfigurable Computing

Lecture 16

Reconfigurable Coprocessors

Overview

- Differences between reconfigurable coprocessors, reconfigurable functional units, and soft processors
- Motivation
- Compute Models: how to fit into computation
- Interaction with memory is a key
- Acknowledgment: DeHon

Overview

- Processors efficient at sequential codes, regular arithmetic operations.
- FPGA efficient at fine-grained parallelism, unusual bit-level operations.
- Tight-coupling important: allows sharing of data/control
- Efficiency is an issue:
  - Context-switches
  - Memory coherency
  - Synchronization

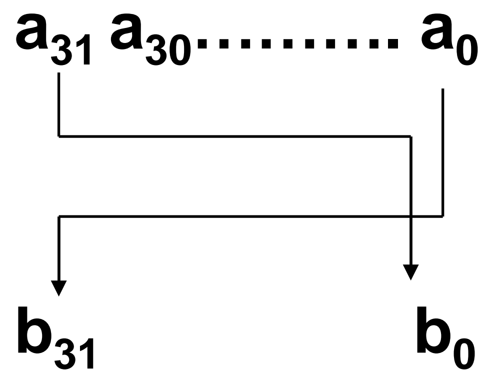
Compute Models

- I/O pre/post processing
- Application specific operation
- Reconfigurable Co-processors
  - Coarse-grained
  - Mostly independent
- Reconfigurable Functional Unit
  - Tightly integrated with processor pipeline
  - Register file sharing becomes an issue

Instruction Augmentation

- Processor can only describe a small number of basic computations in a cycle
  - I bits -> 2^I operations
- Many operations could be performed on 2 W-bit words.
- ALU implementations restrict execution of some simple operations.
  - e. g. bit reversal

Swap bit positions



Instruction Augmentation

- Provide a way to augment the processor instruction set for an application.
- Avoid mismatch between hardware/software

What's Required?

- Fit augmented instructions into data and control stream.
- Create a functional unit for augmented instructions.
- Compiler techniques to identify/use new functional unit.

PRISC

- **Architecture:**
  - **couple into register file as "superscalar" functional unit**
  - **flow-through array (no state)**

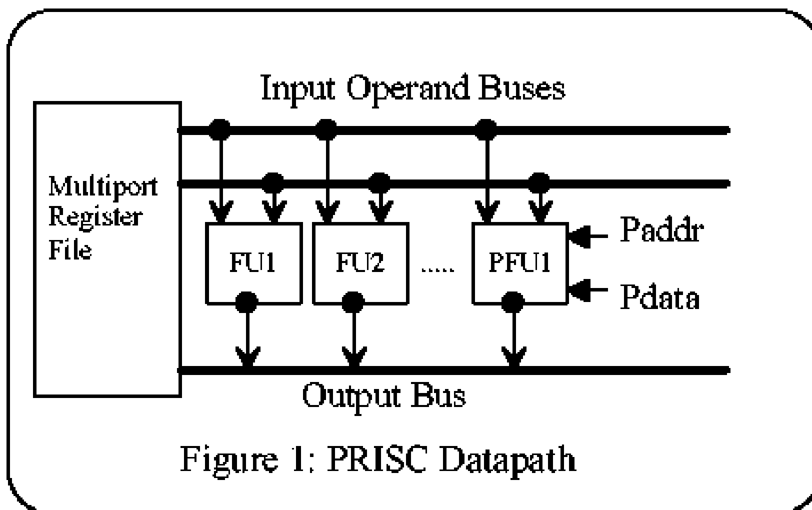
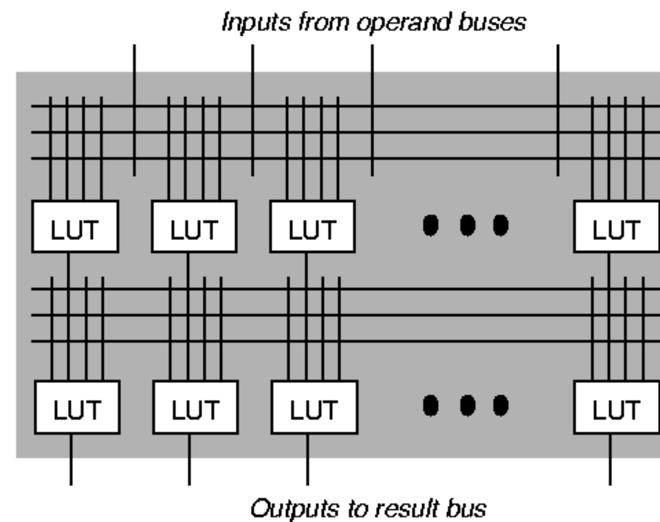


Figure 1: PRISC Datapath



PRISC Results

- All compiled
- working from MIPS binary
- <200 4LUTs ?
  - 64x3
- 200MHz MIPS base

Optimization	CPS	EQN	EXP	GCC	L1	SC
PFU-expression	9	0	48	13	4	12
PFU-table-lookup	0	0	0	0	0	0
PFU-predication	0	1	0	13	0	0
PFU-jump	10	0	47	103	0	35
PFU-loop	0	3	0	4	0	0
TOTAL	19	4	95	133	4	47

Table 1: Static PFU optimization instances in SPECint92.

	CPS	EQN	EXP	GCC	L1	SC
Speedup	1.15	1.91	1.16	1.10	1.06	1.12

Table 2: Cycle count speedup for a PRISC-1 microarchitecture with a single PFU resource. The speedup for each application is an arithmetic average (as defined by SPEC) of all of the data sets for that application.

Razdan/Micro27

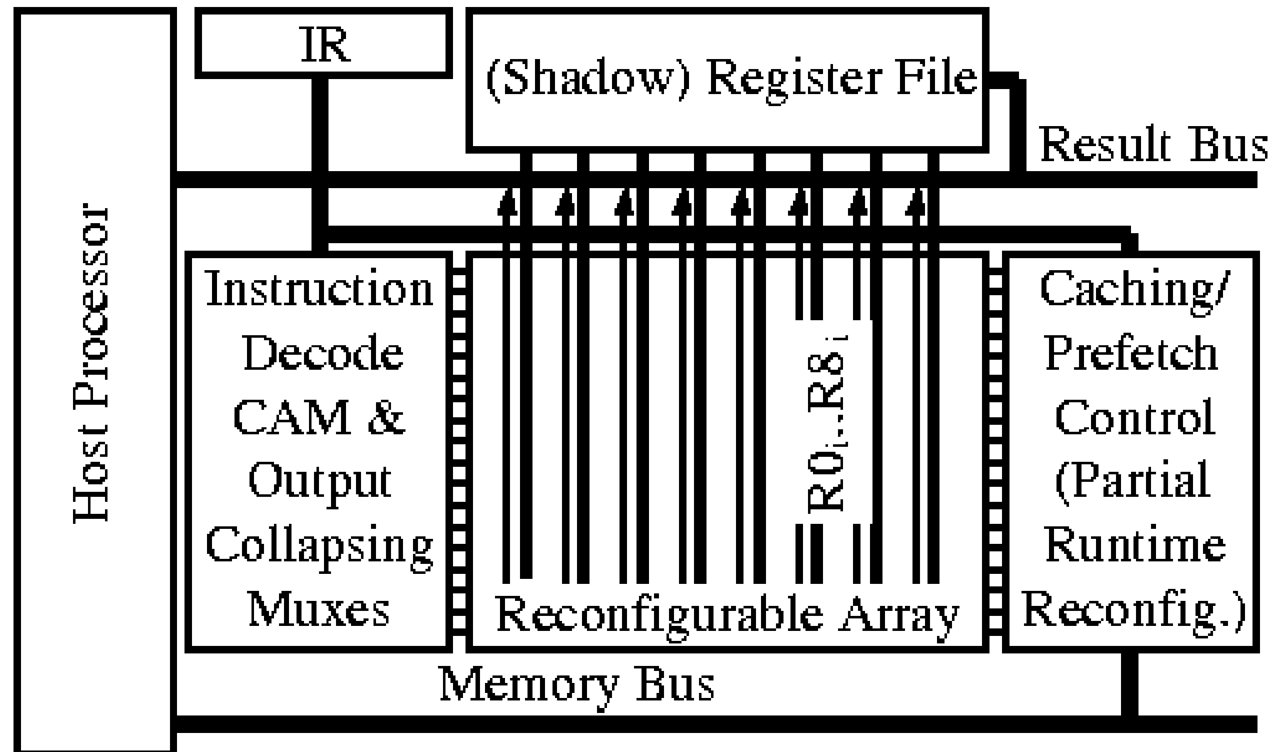
Chimaera

- Start from Prisc idea.
  - Integrate as a functional unit
  - No state
  - RFU Ops (like expfu)
  - Stall processor on instruction miss

- Add
  - Multiple instructions at a time
  - More than 2 inputs possible

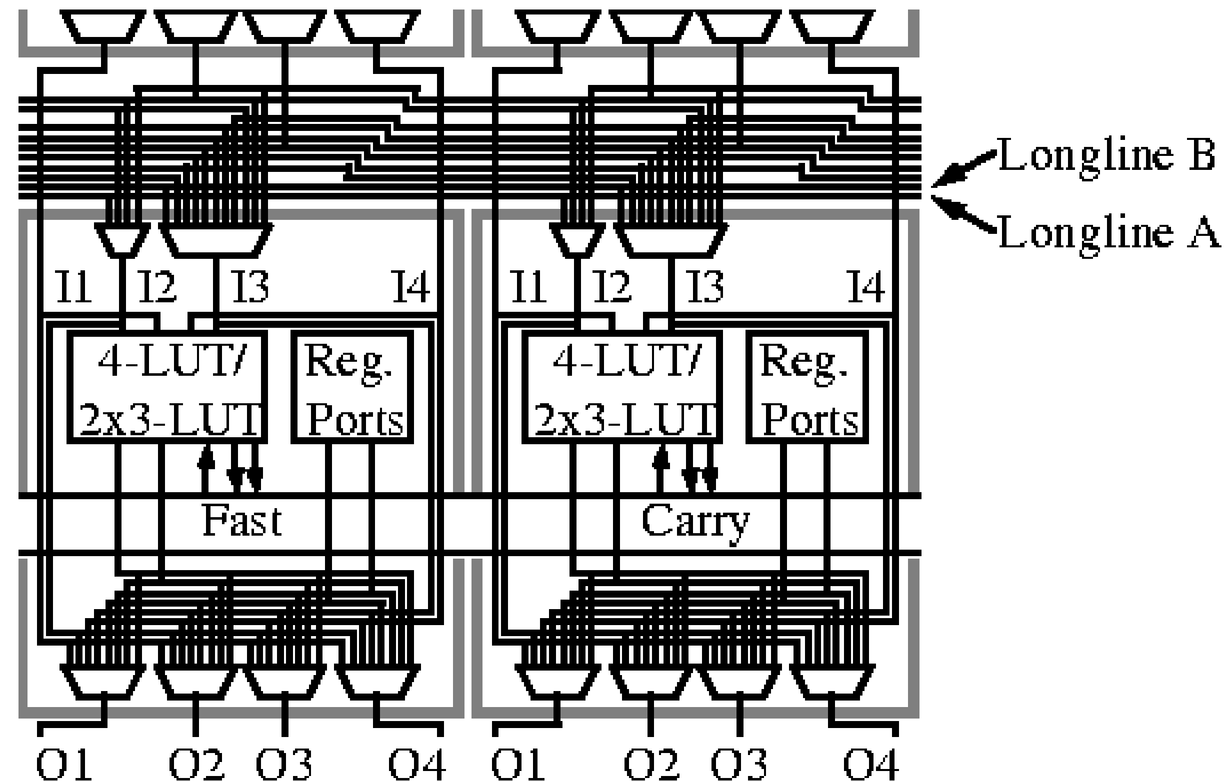
- Hauck: University of Washington

Chimaera Architecture



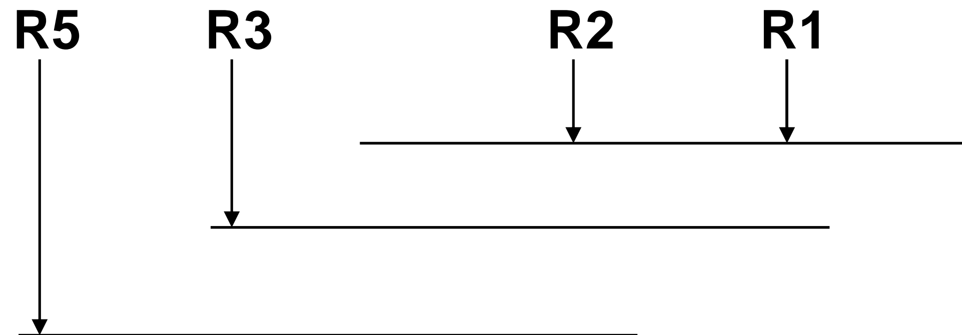
- Live copy of register file values feed into array
- Each row of array may compute from register of intermediates
- Tag on array to indicate RFUOP

Chimaera Architecture



- Array can operate on values as soon as placed in register file.
- Logic is combinational
- When RFUOP matches
  - Stall until result ready
  - Drive result from matching row

Chimaera Timing



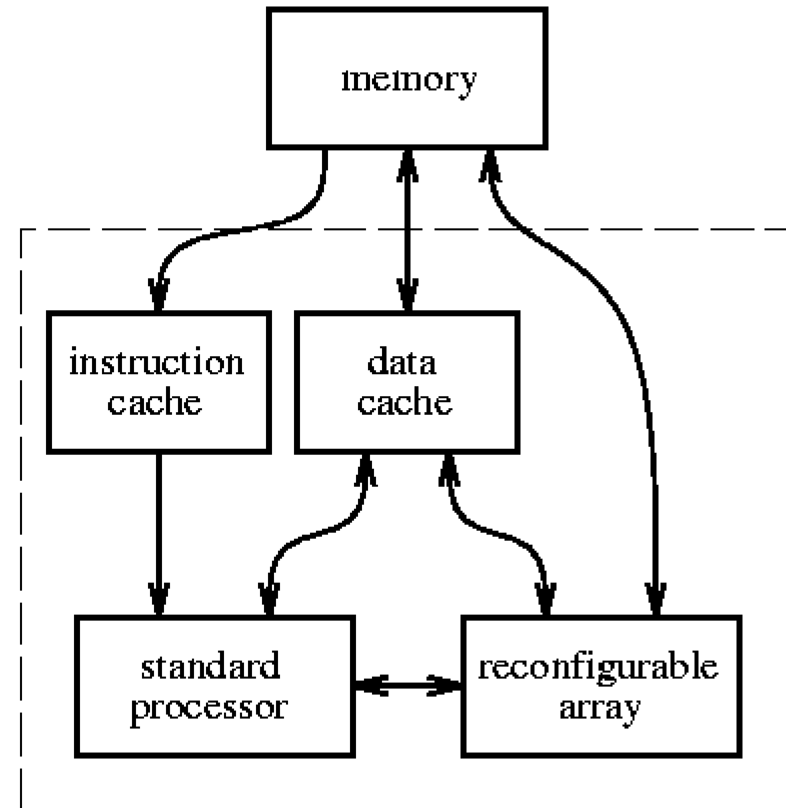
- If R1 presented late then stall
- Might be helped by instruction reordering
- Physical implementation an issue.
- Relies on considerable processor interaction for support

Chimaera Results

- Three Spec92 benchmarks
  - Compress 1.11 speedup
  - Eqntott 1.8
  - Life 2.06
- Small arrays with limited state
- Small speedup
- Perhaps focus on global router rather than local optimization.

Garp

- Integrate as coprocessor
  - Similar bandwidth to processor as functional unit
  - Own access to memory
- Support multi-cycle operation
  - Allow state
  - Cycle counter to track operation
- Configuration cache, path to memory



Garp – UC Berkeley

- ISA – coprocessor operations
  - Issue gaconfig to make particular configuration present.
  - Explicitly move data to/from array
  - Processor suspension during coproc operation
  - Use cycle counter to track progress

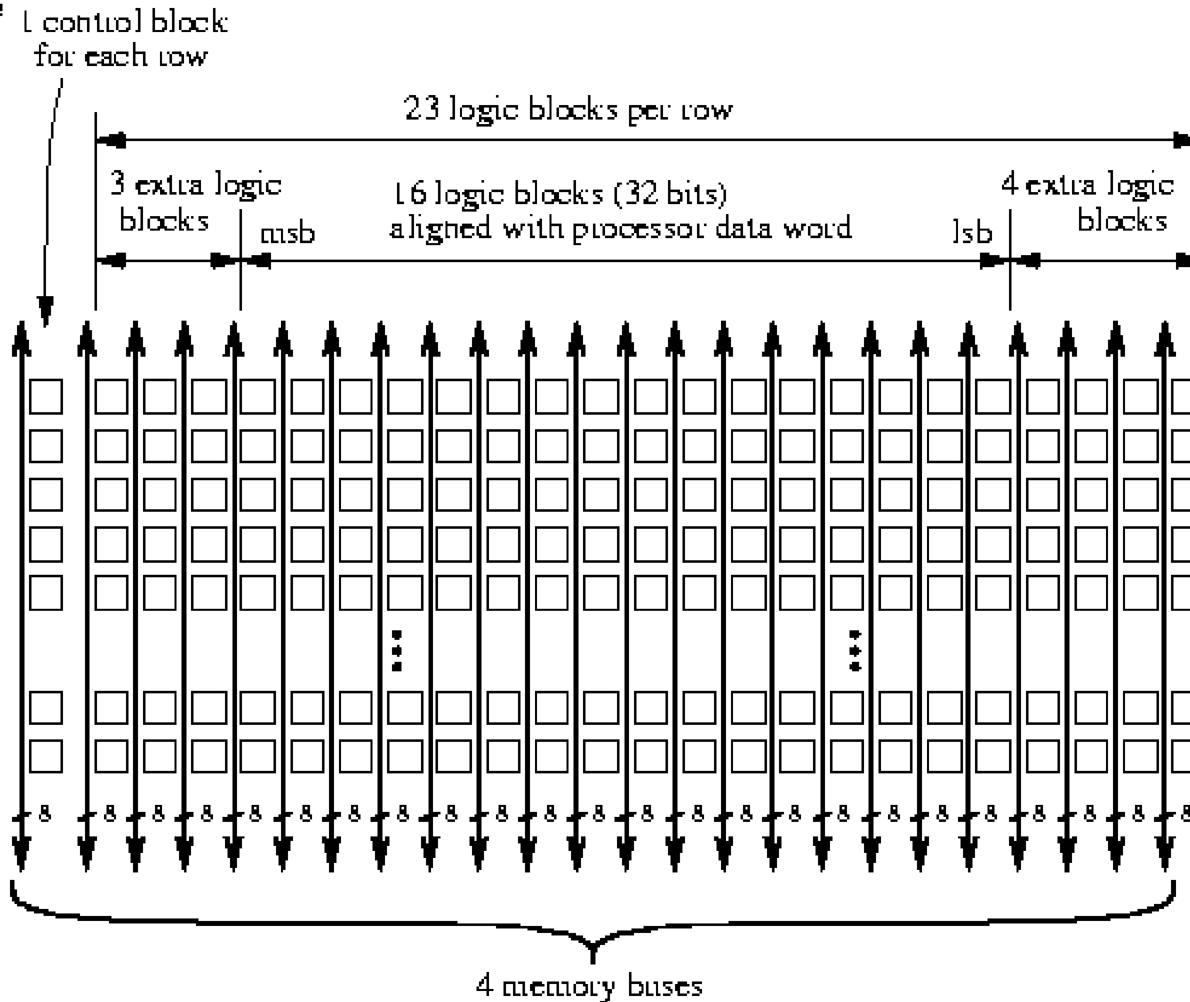
- Array may directly access memory
  - Processor and array share memory
  - Exploits streaming data operations
  - Cache/MMU maintains data consistency

Garp Instructions

Instruction	Interlock?	Description
`gaconf` *reg*	yes	Load (or switch to) configuration at address given by *reg*.
`mtga` *reg, array-row-reg, count*	yes	Copy *reg* value to *array-row-reg* and set array clock counter to *count*.
`mfga` *reg, array-row-reg, count*	yes	Copy *array-row-reg* value to *reg* and set array clock counter to *count*.
`gabump` *reg*	no	Increase array clock counter by value in *reg*.
`gastop` *reg*	no	Copy array clock counter to *reg* and stop array by zeroing clock counter.
`gacinv` *reg*	no	Invalidate cache copy of configuration at address given by *reg*.
`cfga` *reg, array-control-reg*	no	Copy value of array control register *array-control-reg* to *reg*.
`gasave` *reg*	yes	Save all array data state to memory at address given by *reg*.
`garestore` *reg*	yes	Restore previously saved data state from memory at address given by *reg*.

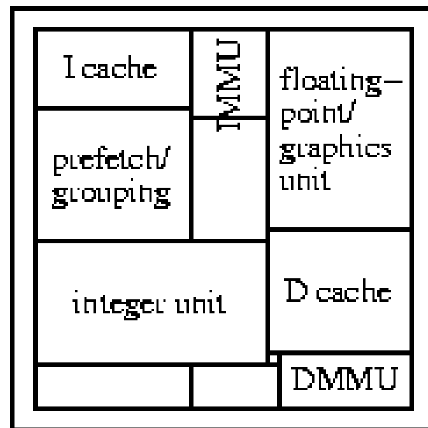
- **Interlock indicates if processor waits for array to count to zero.**
- **Last three instructions useful for context swap**
- **Processor decode hardware augmented to recognize new instructions.**

Garp Array

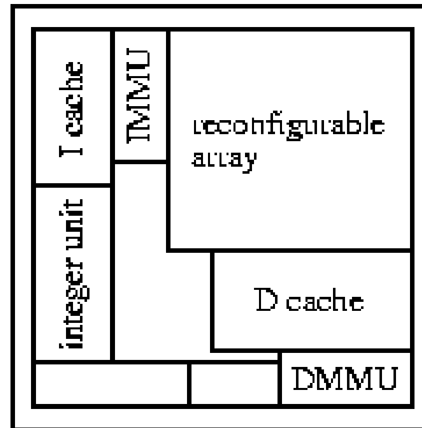


- Row-oriented logic
- Dedicated path for processor/memory
- Processor does not have to be involved in array-memory path

Garp Results



(a) UltraSPARC.



(b) Hypothetical Garp.

Figure 13: Floorplan of the UltraSPARC die, and that of a hypothetical Garp die constructed in the same technology.

Benchmark	167 MHz SPARC	133 MHz Garp	ratio
DES encrypt of 1 MB	3.60 s	0.15 s	24
Dither of 640 × 480 image	160 ms	17 ms	9.4
Sort of 1 million records	1.44 s	0.67 s	2.1

Figure 14: Benchmark results. The times for Garp are obtained from program simulation.

- **General results**
  - **10-20X improvement on stream, feed-forward operation**
  - **2-3x when data dependencies limit pipelining**
  - **[Hauser-FCCM97]**

PRISC/Chimaera vs. Garp

- **Prisc/Chimaera**
  - Basic op is single cycle: <u>expfu</u>
  - No state
  - Could have multiple PFUs
  - Fine grained parallelism
  - Not effective for deep pipelines

- **Garp**
  - Basic op is multi-cycle – gaconfig
  - Effective for deep pipelining
  - Single array
  - Requires state swapping consideration

Common Theme

- To overcome instruction expression limits:
  - Define new array instructions. Make decode hardware slower / more complicated.
  - Many bits of configuration… swap time. An issue -> recall tips for dynamic reconfiguration.
- Give array configuration short "name" which processor can call out.
- Store multiple configurations in array. Access as needed (DPGA)

Observation

- All coprocessors have been single-threaded
  - Performance improvement limited by application parallelism
- Potential for task/thread parallelism
  - DPGA
  - Fast context switch
- Concurrent threads seen in discussion of IO/stream processor
- Added complexity needs to be addressed in software.

Parallel Computation: Processor and FPGA

- What would it take to let the processor and FPGA run in parallel?

Modern Processors

Deal with:

- Variable data delays
- Dependencies with data
- Multiple heterogeneous functional units

Via:

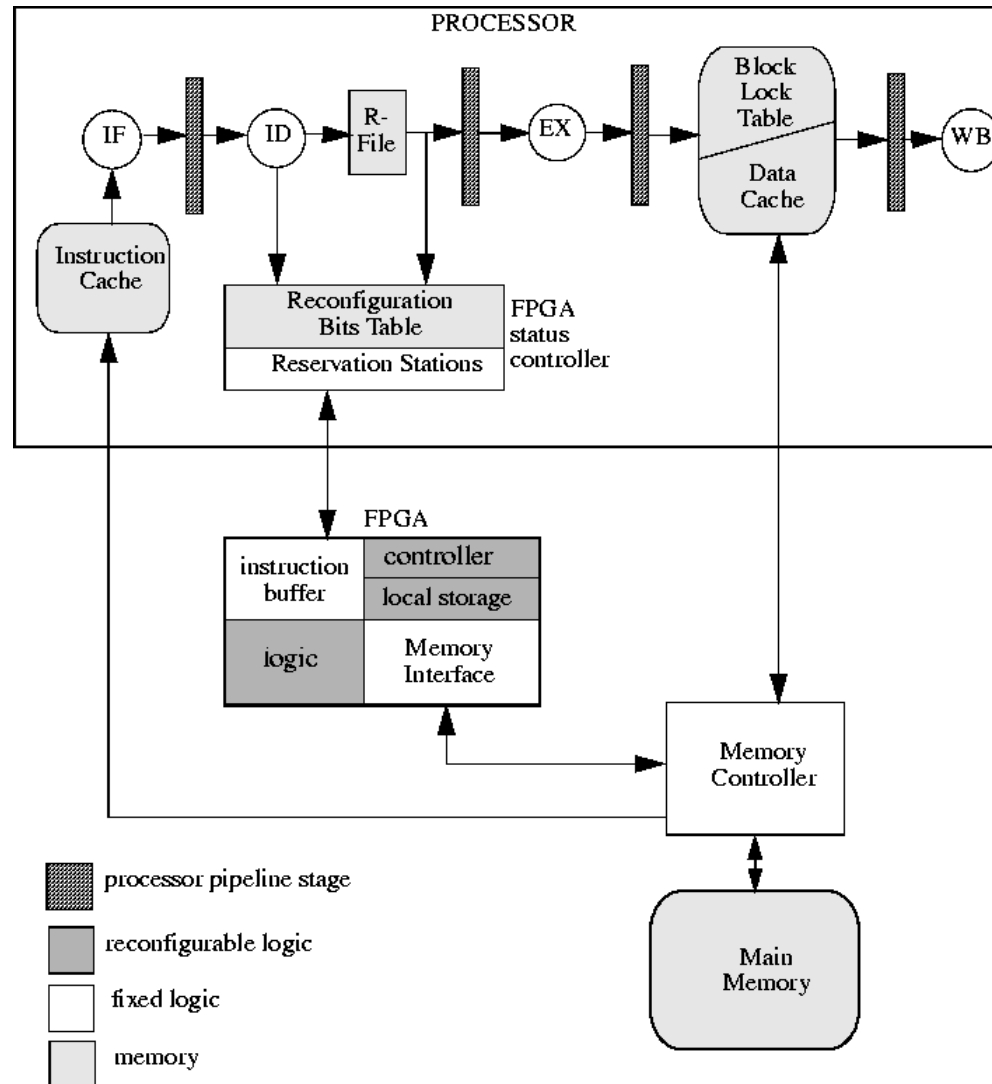
- Register scoreboarding
- Runtime data flow (Tomasulo)

OneChip

- Want array to have direct memory→memory operations
- Want to fit into programming model/ISA
  - w/out forcing exclusive processor/FPGA operation
  - allowing decoupled processor/array execution
- Key Idea:
  - FPGA operates on memory→memory regions
  - make regions explicit to processor issue
  - scoreboard memory blocks

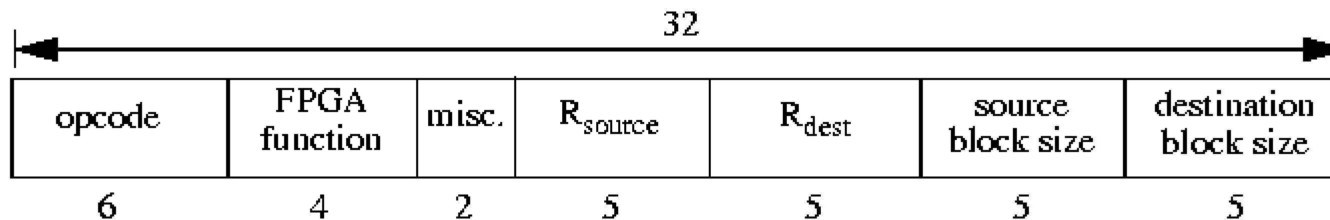
[Jacob+Chow: Toronto]

OneChip Pipeline



OneChip Instructions

- **Basic Operation is:**
  - FPGA MEM[Rsource]→MEM[Rdst]
    - block sizes powers of 2



- **Supports 14 "loaded" functions**
  - DPGA/contexts so 4 can be cached

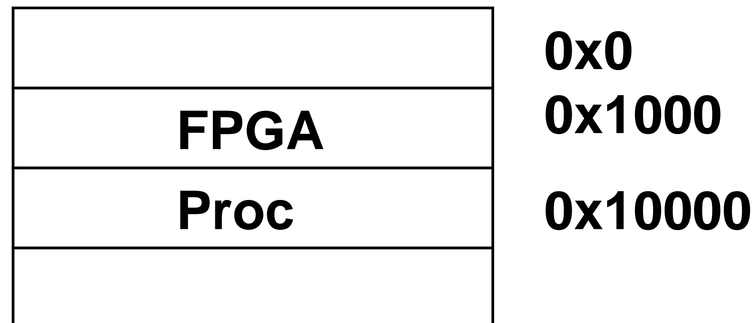
- **Fits well into soft-core processor model**

OneChip

- Basic op is: FPGA MEM→MEM
- no state between these ops
- coherence is that ops appear sequential
- could have multiple/parallel FPGA Compute units
  - scoreboard with processor and each other
- single source operations?
- can't chain FPGA operations?

OneChip Extensions

- FPGA operates on certain memory regions only
- Makes regions explicit to processor issue.
- Scoreboard memory blocks



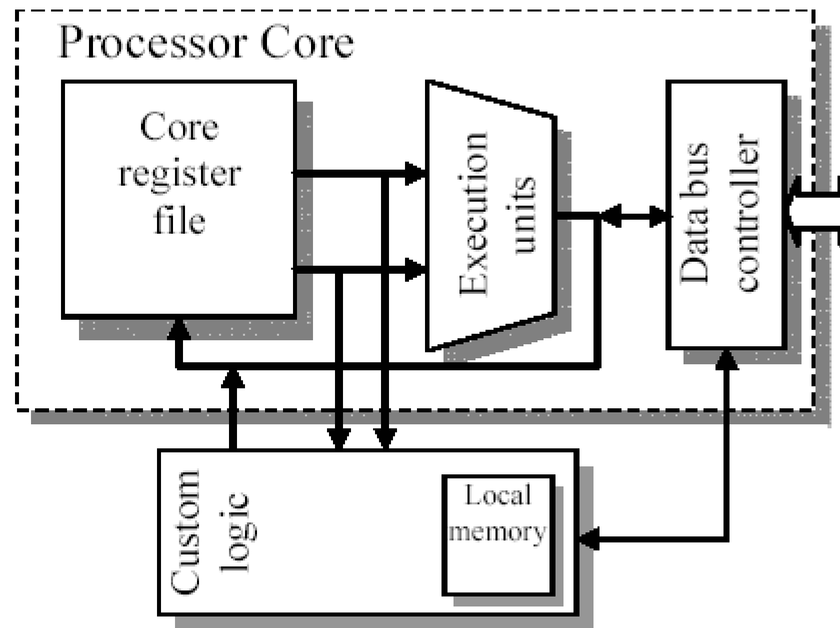
Indicates usage of data pages like virtual memory system!

Model Roundup

- Interfacing
- IO Processor (Asynchronous)
- Instruction Augmentation
  - PFU (like FU, no state)
  - Synchronous Coproc
  - VLIW
  - Configurable Vector
- Asynchronous Coroutine/coprocesor
- Memory⇒memory coprocessor

Shadow Registers for Functional Units

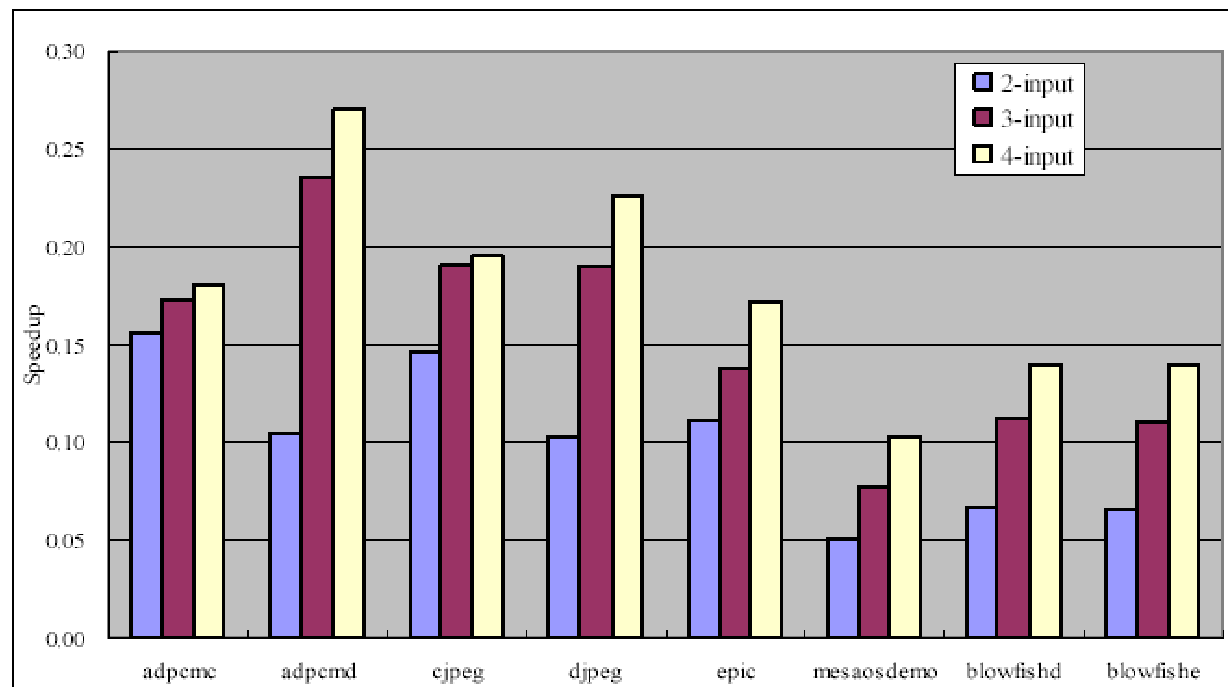
- Reconfigurable functional units require tight integration with register file
- Many reconfigurable operations require more than two operands at a time



Cong: 2005

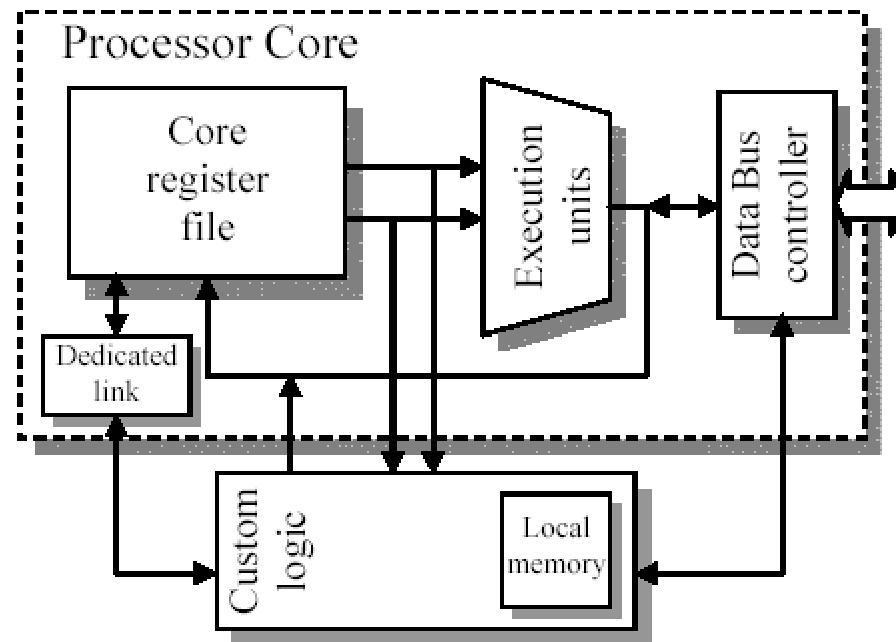
Multi-operand Operations

- What's the best speedup that could be achieved?
  - Provides upper bound
- Assumes all operands available when needed



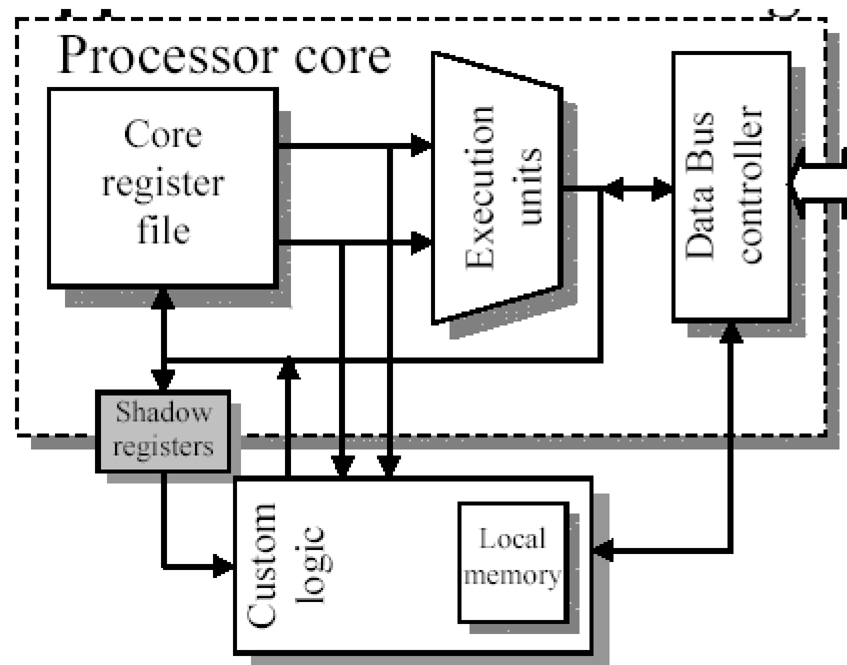
Providing Additional Register File Access

- Dedicated link – move data as needed
  - Requires latency
- Extra register port – consumes resources
  - May not be used often
- Replicate whole (or most) of register file
  - Can be wasteful

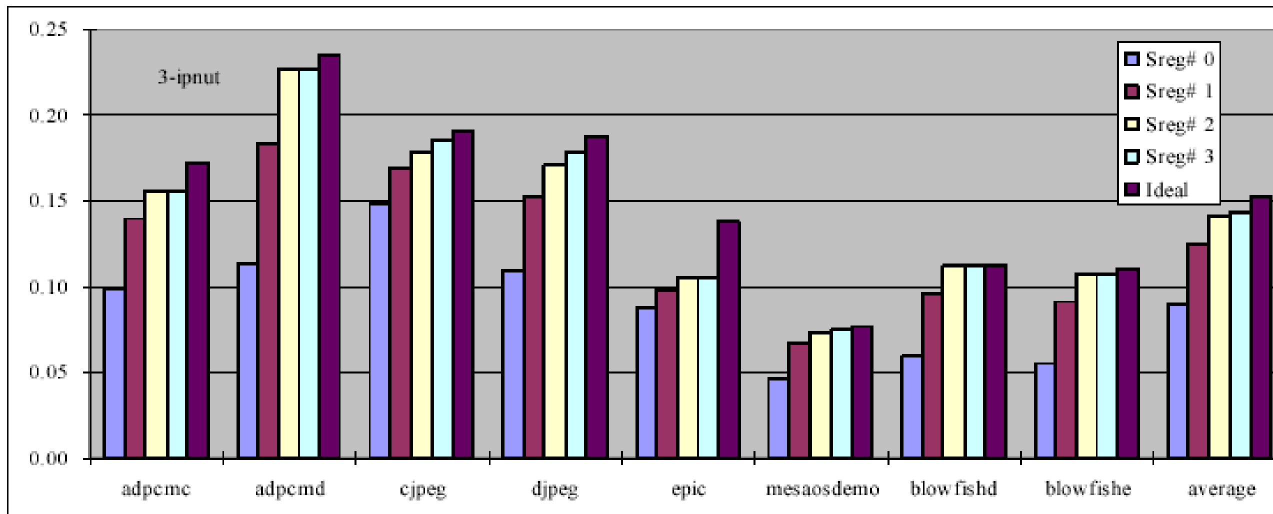


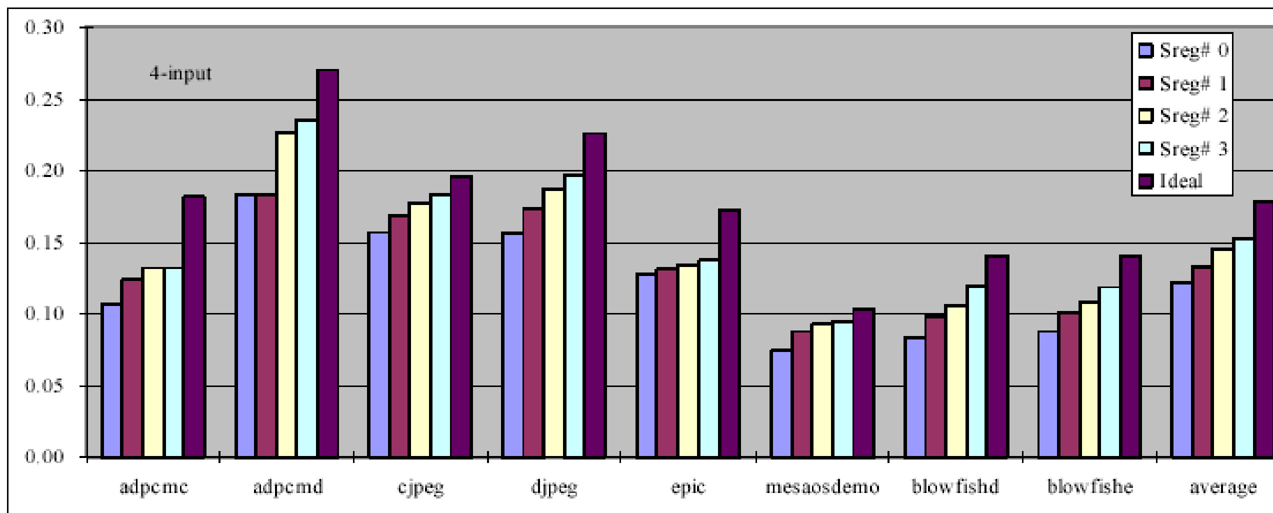
Shadow Register Approach

- Small number of registers needed (3 or 4)
- Use extra bits in each instruction
- Can be scaled for necessary port size



Shadow Register Approach





- Approach comes within 89% of ideal for 3-input functions
- Paper also shows supporting algorithms

Summary

- Many different models for co-processor implementation
  - Functional unit
  - Stand-alone co-processor
- Programming models for these systems is a key
- Recent compiler advancements open the door for future development
- Need tie in with applications