Exploring Parallel Out-of-Order Re-execution

David Morano
David Kaeli

Northeastern University
Computer Architecture Research

BARC 04/01/30

outline

- Introduction
- Execution window
- Basic machine operation
- Issue stations
- Example execution
- Summary

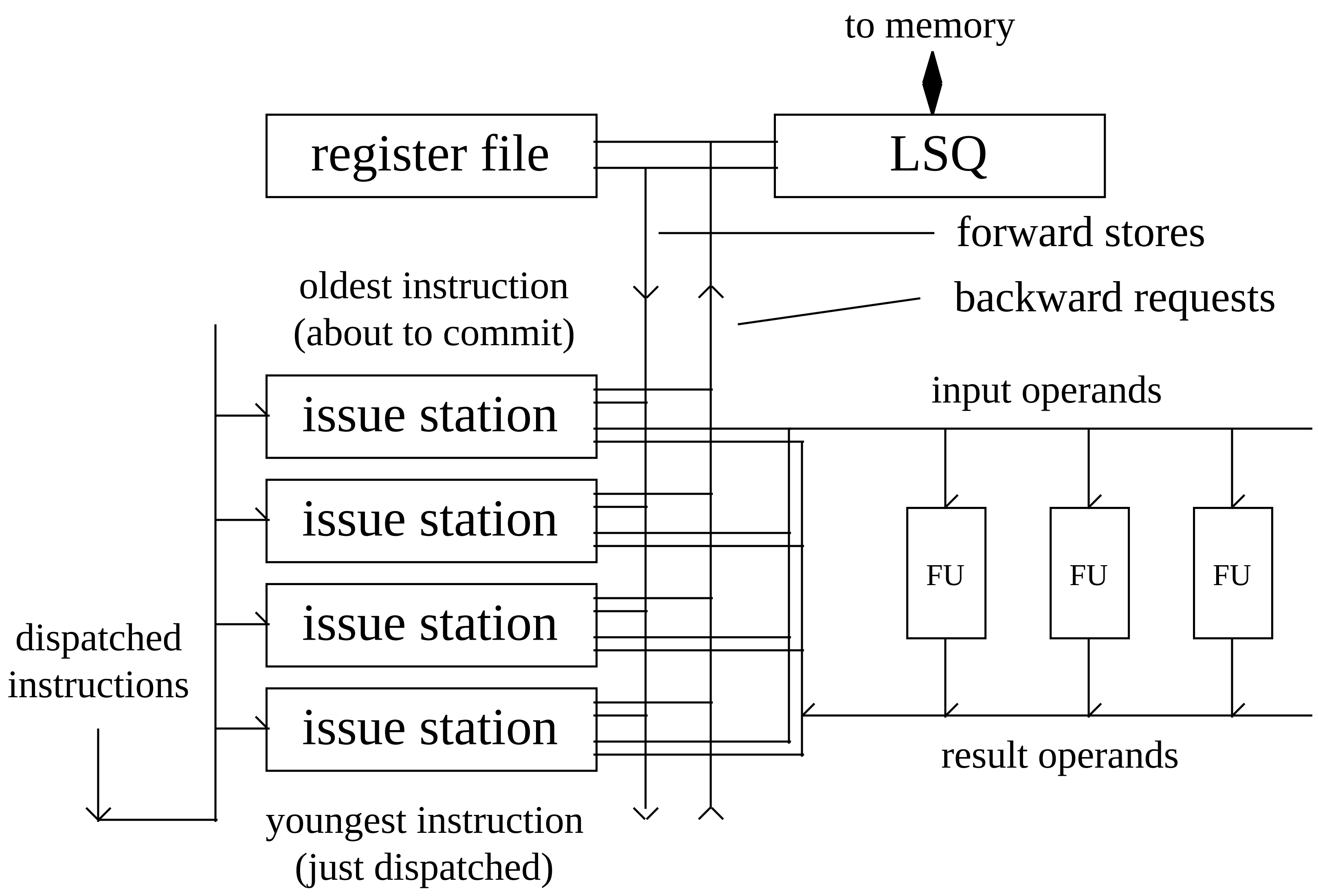
introduction

- **Attempting to take advantage of previous work that has shown some promise in extracting ILP from sequential programs**
  - IPC of about 6 for 256 simultaneously re-executing instructions
  - IPC of about 7.5 for 512 simultaneously re-executing instructions
- **Applying some of the re-execution ideas to a more conventional superscalar microarchitecture**
- **Some ideas to be carried forward :**
  - retaining binary program compatibility to existing ISAs
  - breaking control and data dependencies
  - time-ordering-tags as the dependency enforcement mechanism
  - designing for re-executions of all instruction types (not just memory loads)
  - dynamic handling of speculative execution and operand management

major components

- **Issue stations**
  - similar to a reservation station
  - holds instruction operation until ready for retirement
  - the instruction operation "issues" from this structure to an available function unit when needed
  - combines some of the functions of an issue window, RUU, and ROB into a common structure
- **Function units**
  - generally the same as existing ones
  - returns result operands to the originating issue station rather than writing results to an RUU or ROB
- **Register file**
  - not needed for renaming or speculative results
- **Familiar load-store queue and memory hierarchy**

execution window

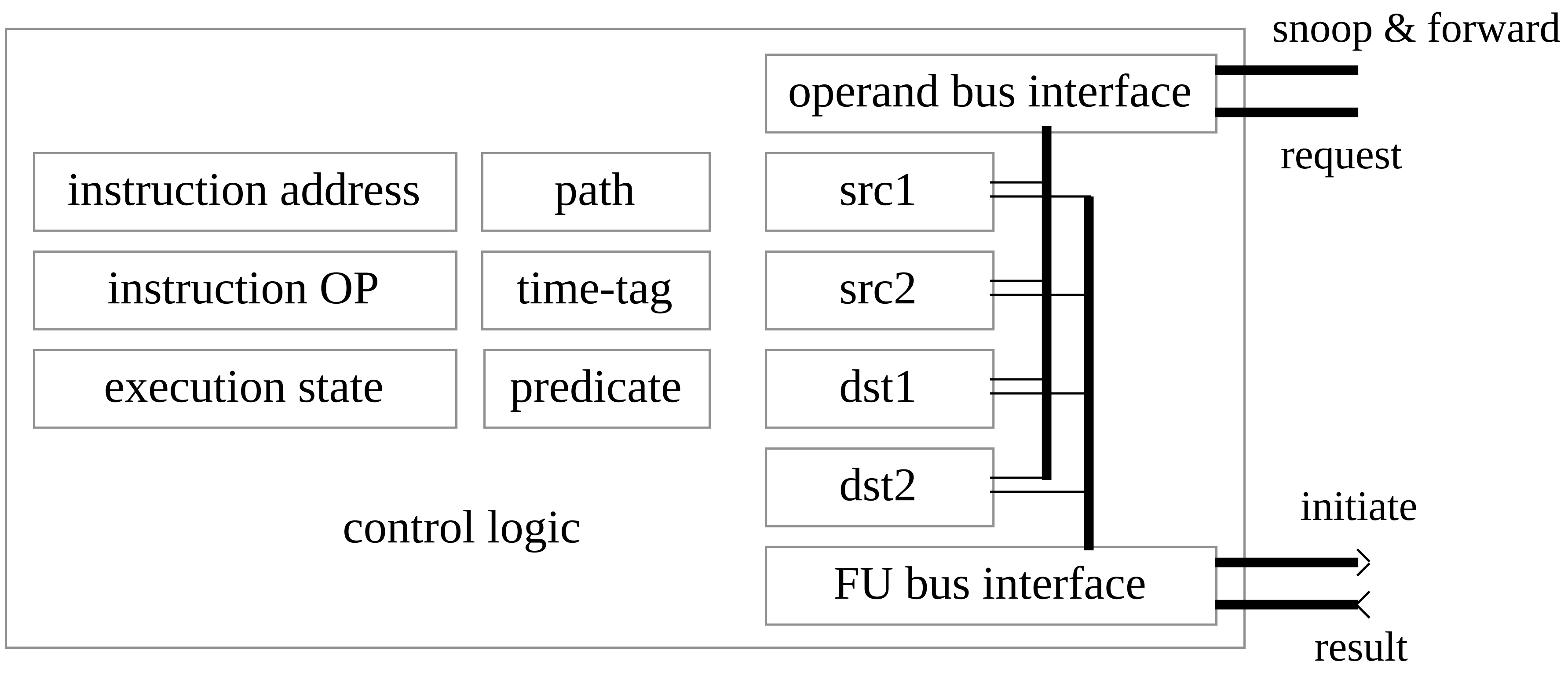


basic operation

- Instructions are decoded at fetch time and stored in fetch buffers
- Decoded instructions are dispatched to Issue Stations (IS)
- ISes contend with each other for a FU resource (waiting as needed)
- ISes send an operation along with its input operands to a FU when available
- The IS waits for the FU execution result
- Resulting operand returns to the originating IS
- IS forwards the result operand to other ISes in program-ordered future
- ISes who snarf new (different) input operands proceed to re-execute as needed

issue station

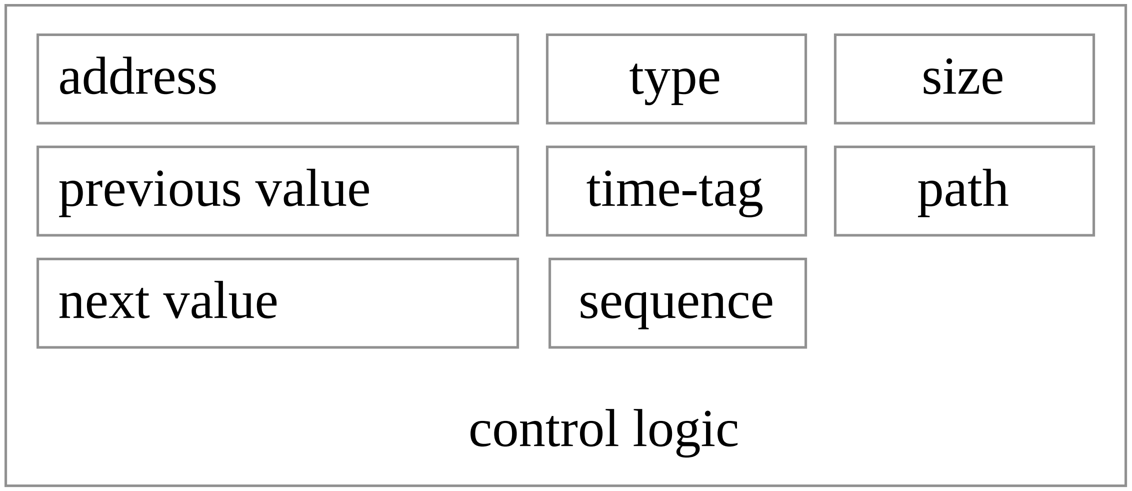
- Similar to reservation station
- Implements dynamic operand renaming (for registers and memory)



- Two input and two output operands are shown (varies w/ ISA)

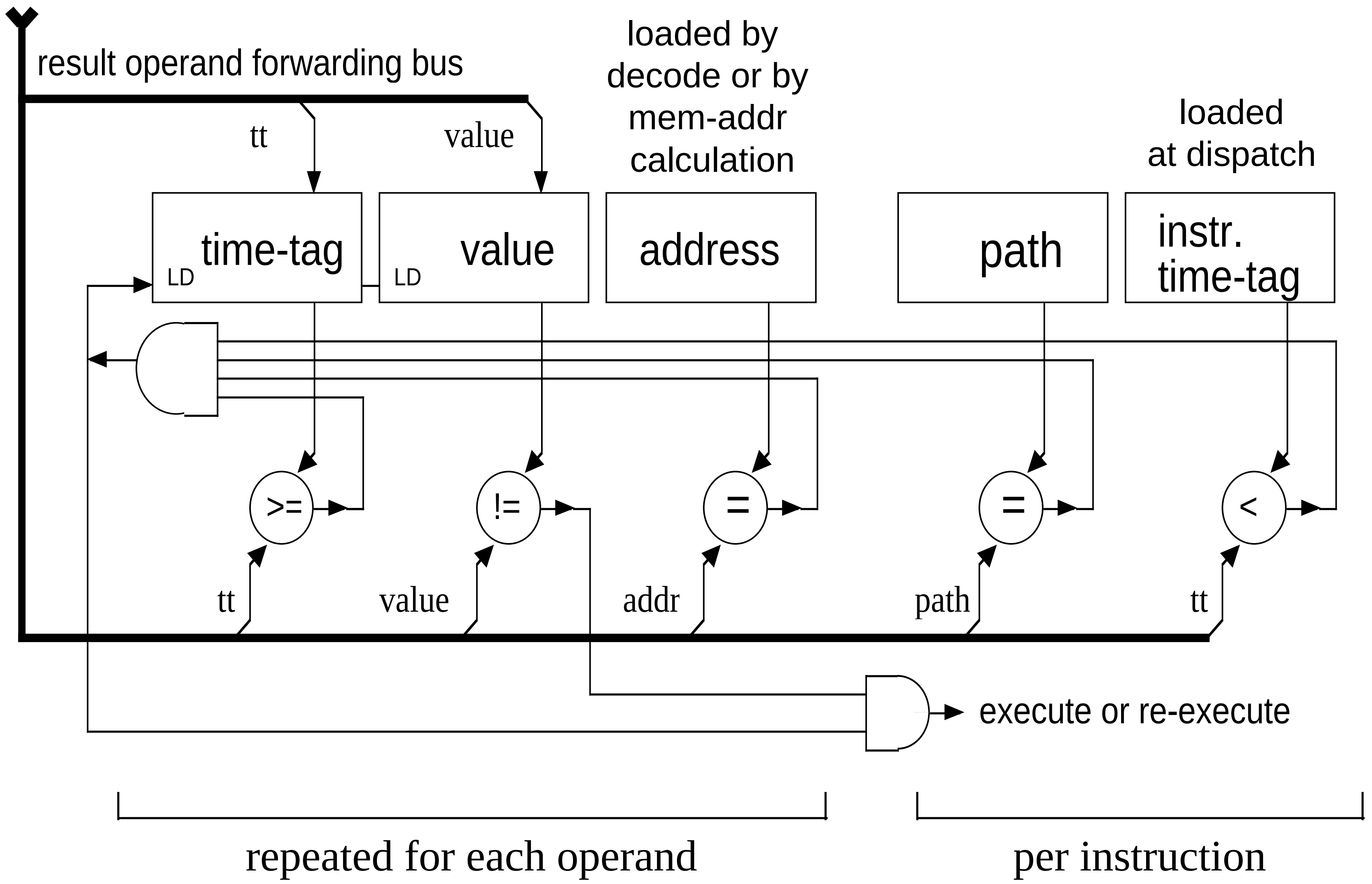
operand state block

- Holds all information about one operand
- Includes necessary logic to snoop for updates



- Operand names take the form -> type : path : time-tag : seq : addr
- Example name for a register -> "register : 1 : 27 : 3 : 6"
- Predicates are operands also but have additional state (not shown)

snoop/snarf operation



some additional IS state

- – Acquiring input operands
- – Execution is needed (waiting for FU availability)
- – Executing (waiting for FU result to return)
- – Executed at least once
- – Result operand was requested by another IS
- – Result operand needs to be forwarded
- – Operand is being forwarded

- Most of these indications also prevent instruction commitment
- Retirement (squash) may still occur under certain conditions

issue station operation

- An intruction gets dispatched to an IS with (choices) :
  - initial input operands from :
    - architected register file
    - from a value predictor
  - no initial input operands
- If input operands are available, arbitrate for FU resource, otherwise acquire input operands by requesting them
- After an execution result is available, "forward" the operand
- Continuously snoop for new input operands
- Initiate (arbitrate for FU resource) execution when a changed input operand arrives
- Respond to requests by other ISes for operands
- Track all in-progress conditions for commitment determination

example execution (registers)

code fragment

label	*TT*	*instruction*
i1	0	r3 <= 1
i2	1	r4 <= r3 + 1
i3	2	r3 <= 2
i4	3	r5 <= r3 + 2

example execution schedule

cycle	*execute*	*forward*	*snarf*
0	i1		
1		i1{r3=1}	
2			i2{r3=1}, i4{r3=1}
3	i2, i4		
4	i3	i2{r4=2}, i4{r5=3}	
5		i3{r3=2}	
6			i4{r3=2}
7	i4		
8		i4{r5=4}	

- we want *r3* to have value =2 after execution of i3
- we want *r5* to have value =4 after execution of i4

- i4 executes in clock 3 after snarfing *r3* from i1, resulting in *wrong* result for *r5*
- i4 executes again after snarfing *r3* from i3, giving correct result for *r5*

summary

- **Proposing a microarchitecture to explore OoO re-execution, but more conventional than our previous designs**
- **Will explore :**
  - the nature and amount of re-execution that may (or may not) be desirable
  - different types and numbers of resources
  - various interconnection topologies, bus fabrics, and bandwidths
- **Microarchitecture can be modified to support a number of hardware mechanisms :**
  - various value prediction techniques
  - dynamic execution-time instruction predication
  - dynamically finding and executing control-independent instructions beyond branch joins