15-740/18-740 Computer Architecture Lecture 23: Superscalar Processing (III)

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Homework 4
  - Out today
  - Due November 15

- Midterm II
  - November 22

- Project Poster Session
  - December 9 or 10

Readings

- Required (New):
  - Patel et al., "Evaluation of design options for the trace cache fetch mechanism," IEEE TC 1999.
  - Palacharla et al., "Complexity Effective Superscalar Processors," ISCA 1997.
- Required (Old):
  - **Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proc. IEEE, Dec. 1995.**
  - Stark, Brown, Patt, "On pipelining dynamic instruction scheduling logic," MICRO 2000.
  - Boggs et al., "The microarchitecture of the Pentium 4 processor," Intel Technology Journal, 2001.
  - Kessler, "The Alpha 21264 microprocessor," IEEE Micro, March-April 1999.
- Recommended:
  - Rotenberg et al., "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," MICRO 1996.

Superscalar Processing

- Fetch (supply N instructions)
- Decode (generate control signals for N instructions)
- Rename (detect dependencies between N instructions)
- Dispatch (determine readiness and select N instructions to execute in-order or out-of-order)
- Execute (have enough functional units to execute N instructions + forwarding paths to forward results of N instructions)
- Write into Register File (have enough ports to write results of N instructions)
- Retire (N instructions)

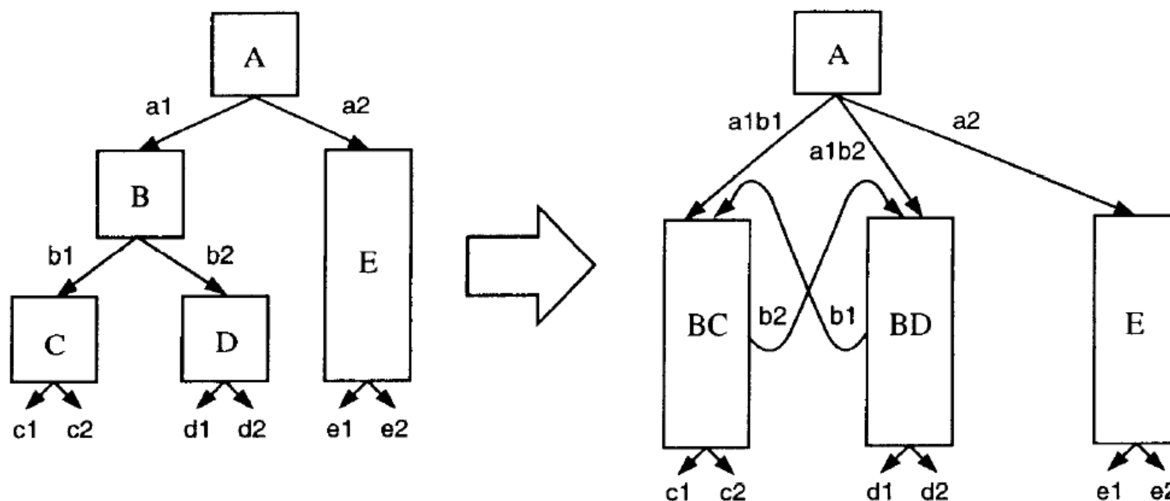
Last Two Lectures

- Superscalar Fetch: Alignment and Fetch Breaks
  - Compiler
    - Code reordering (basic block reordering)
    - Superblock
  - Hardware
    - Split-line fetch (for alignment)
    - Trace cache
  - Hardware/software cooperative
    - Block structured ISA

- Superscalar Decode
  - Pre-decoding
  - Decode cache
  - CISC to RISC translation
  - Instruction buffering (fetch-decode decoupling)

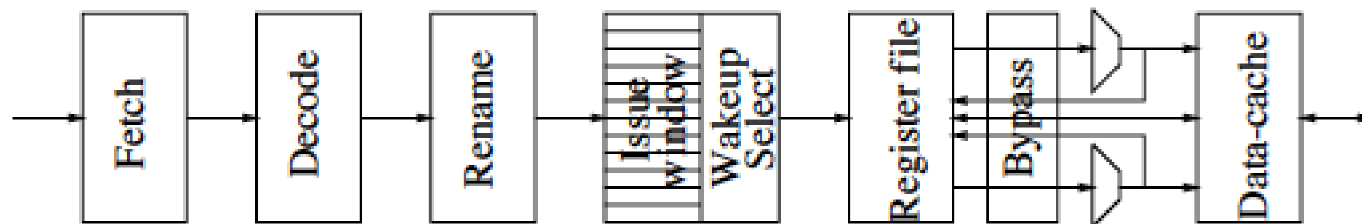
Block Structured ISA

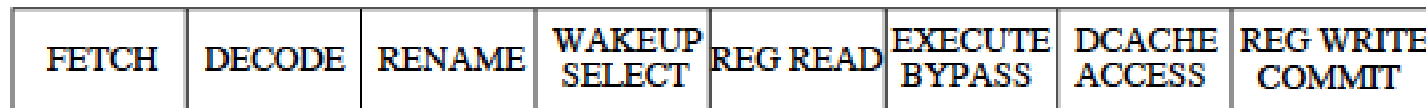
- Blocks (> instructions) are atomic (all-or-none) operations
  - Either all of the block is committed or none of it
- Compiler enlarges blocks by combining basic blocks with their control flow successors
  - Branches within the enlarged block converted to "fault" operations → if the fault operation evaluates to true, the block is discarded and the target of fault is fetched



Superscalar Processing

- Fetch (supply N instructions)
- Decode (generate control signals for N instructions)
- Rename (detect dependencies between N instructions)
- Dispatch (determine readiness and select N instructions to execute in-order or out-of-order)
- Execute (have enough functional units to execute N instructions + forwarding paths to forward results of N instructions)
- Write into Register File (have enough ports to write results of N instructions)
- Retire (N instructions)





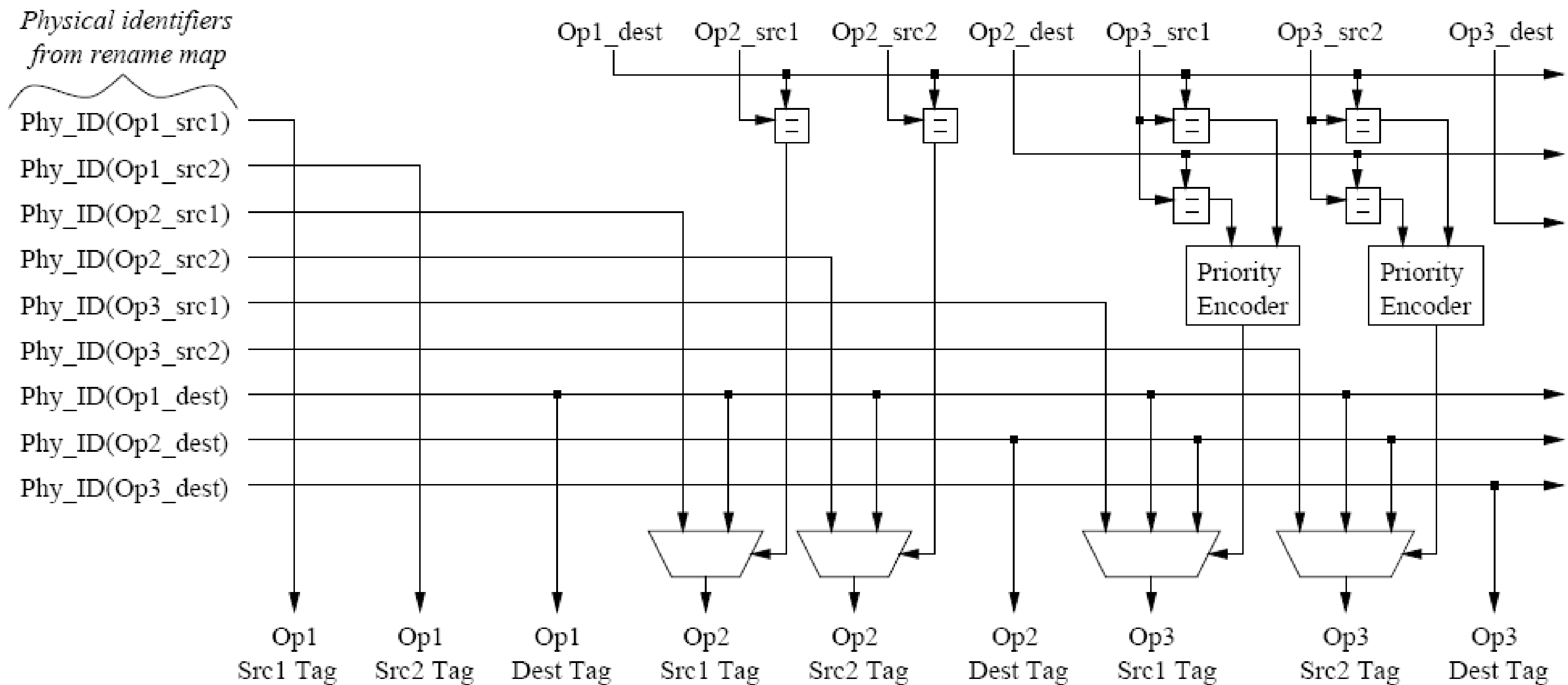
FETCH	DECODE	RENAME	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

Renaming Multiple Instructions per Cycle

- Renaming independent instructions in parallel: easy
- Renaming dependent instructions in the same cycle: harder
- Two issues:
  - Flow dependency: Dependent instruction should get its source tag from a parent instruction renamed in the same cycle
    - Need to compare each arch. source ID with the arch. destination ID of all older instructions
    - N*(N-1) comparators
  - Output dependency: The youngest writer's destination tag should be written into the register alias table

ADD R1 ← R2, R3
ADD R5 ← R1, R1
ADD R1 ← R6, R7

Renaming 3 Instructions in Parallel



- Stark et al., "On pipelining dynamic instruction scheduling logic," MICRO 2000.

Scalability of Register Renaming

- As rename width increases, critical path of dependency check logic (rename logic) increases
  - Need to pipeline rename logic

  -- Increases branch misprediction penalty

    -- Branch misprediction determined 1 cycle later

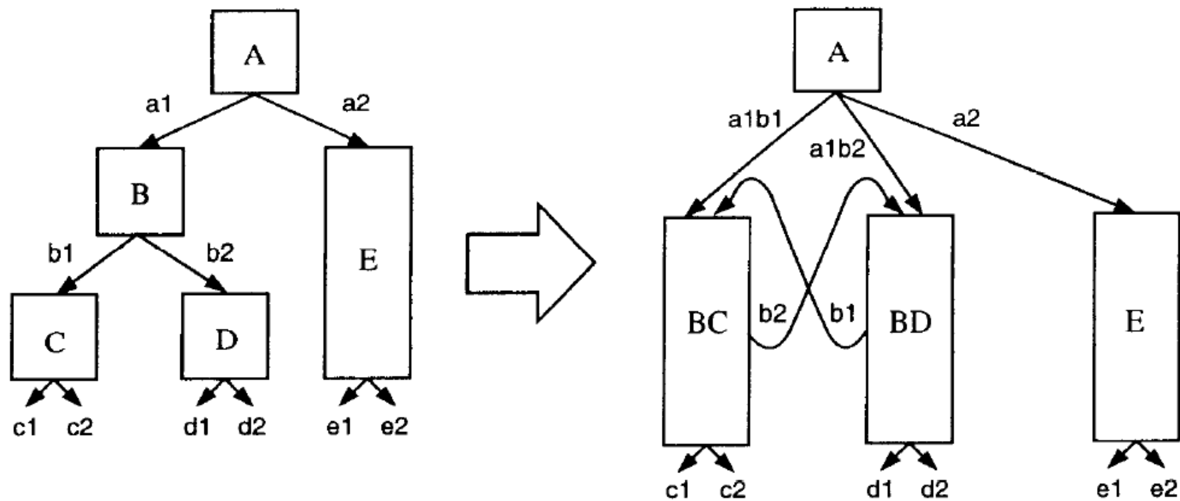
      -- One more wasted pipe stage worth of work

- Idea 1: Compiler ensures all instructions renamed in one cycle are register-independent

  -- Tough to find 2, 3, 4, … independent instructions
- Idea 2: Rename instructions partially at compile time
  - Ensure each instruction in a sequence of instructions writes to a unique register

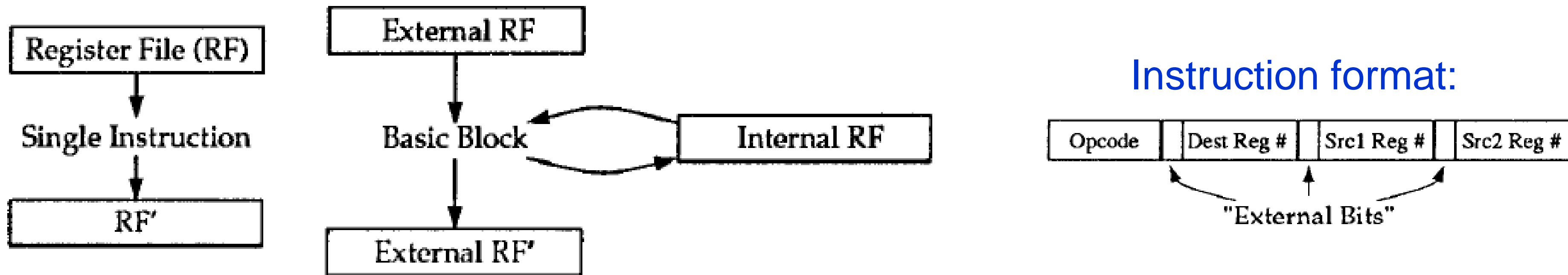
HW/SW Renaming with BS-ISA (I)

- Both ideas can be enhanced if the atomic unit of execution is enlarged
  - Remember the block-structured ISA



- Within a block, ensure each instruction writes to a unique register identifier
  - Tag <= [block ID] @ [unique register ID]

HW/SW Renaming with BS-ISA (II)



- Within-block register communication can be handled using a separate, internal, register file: Tag <= [block ID] @ [unique register ID]
- Only external registers (registers live-out from a block) require renaming and dependency checking

+ No need for dependency check logic

+ Reduced pressure on register file (fewer ports, entries, accesses)

-- ISA changes break backward compatibility

- Sprangle and Patt, "Facilitating superscalar processing via a combined static/dynamic register renaming scheme," MICRO 1994.

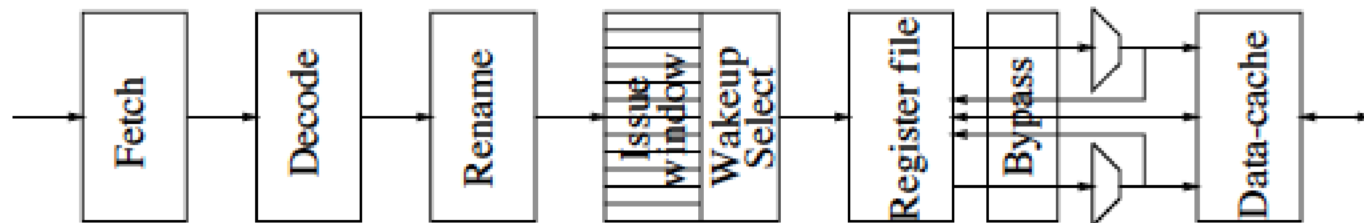
Other Advantages of BS-ISA

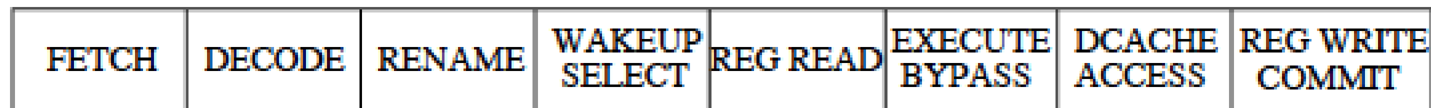
- No ordering needed for instructions within a block
  - Flow dependences explicitly identified
  - No anti or output dependencies

+ Simplifies alignment of instructions within the processor
+ Compiler can order instructions such that
  + Long latency instructions (e.g. loads) executed first
  + Instructions are aligned with functional units they will execute at
+ Compiler can perform aggressive code optimizations that would otherwise be hindered by control flow

Superscalar Processing

- Fetch (supply N instructions)
- Decode (generate control signals for N instructions)
- Rename (detect dependencies between N instructions)
- Dispatch (determine readiness and select N instructions to execute in-order or out-of-order)
- Execute (have enough functional units to execute N instructions + forwarding paths to forward results of N instructions)
- Write into Register File (have enough ports to write results of N instructions)
- Retire (N instructions)





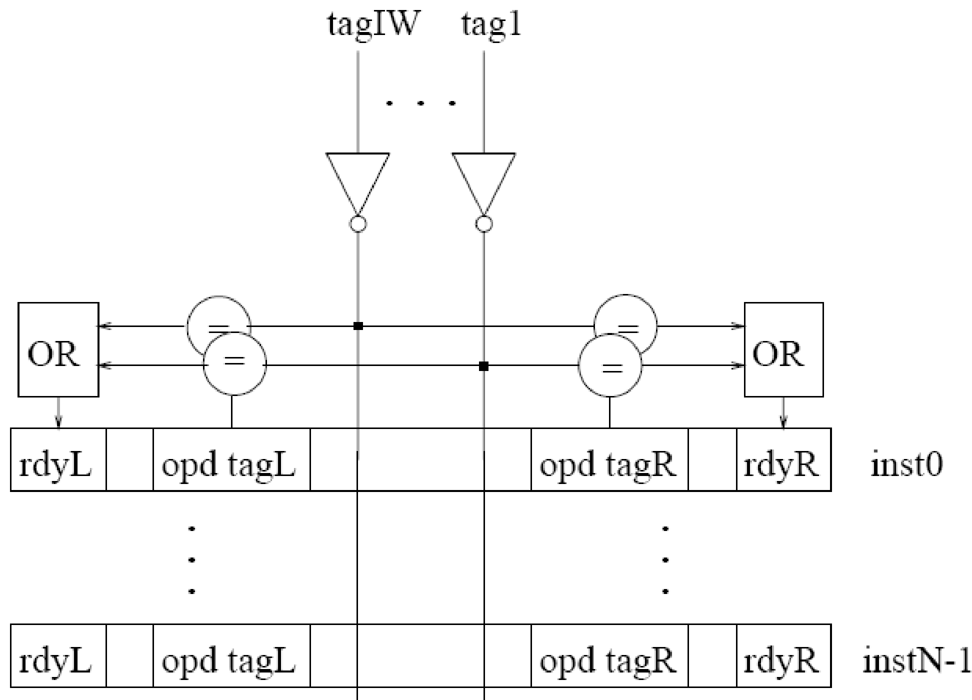
FETCH	DECODE	RENAME	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

Multiple Instruction Dispatch/Scheduling

- Dispatch consists of two operations
  - Wakeup logic: Determining whether an instruction is ready to execute (tracks readiness of all source operands)
  - Selection logic: Choosing instructions for execution from the pool of ready instructions

- Wakeup logic consists of tag matching
  - Content associative matching of all broadcast tags across all instructions in the reservation stations
  - Number of tags broadcast in a cycle = Issue Width (IW)
    - Need IW tag comparators for each source register tag

Wakeup Logic (I)



- Palacharla et al., "Complexity Effective Superscalar Processors," ISCA 1997.

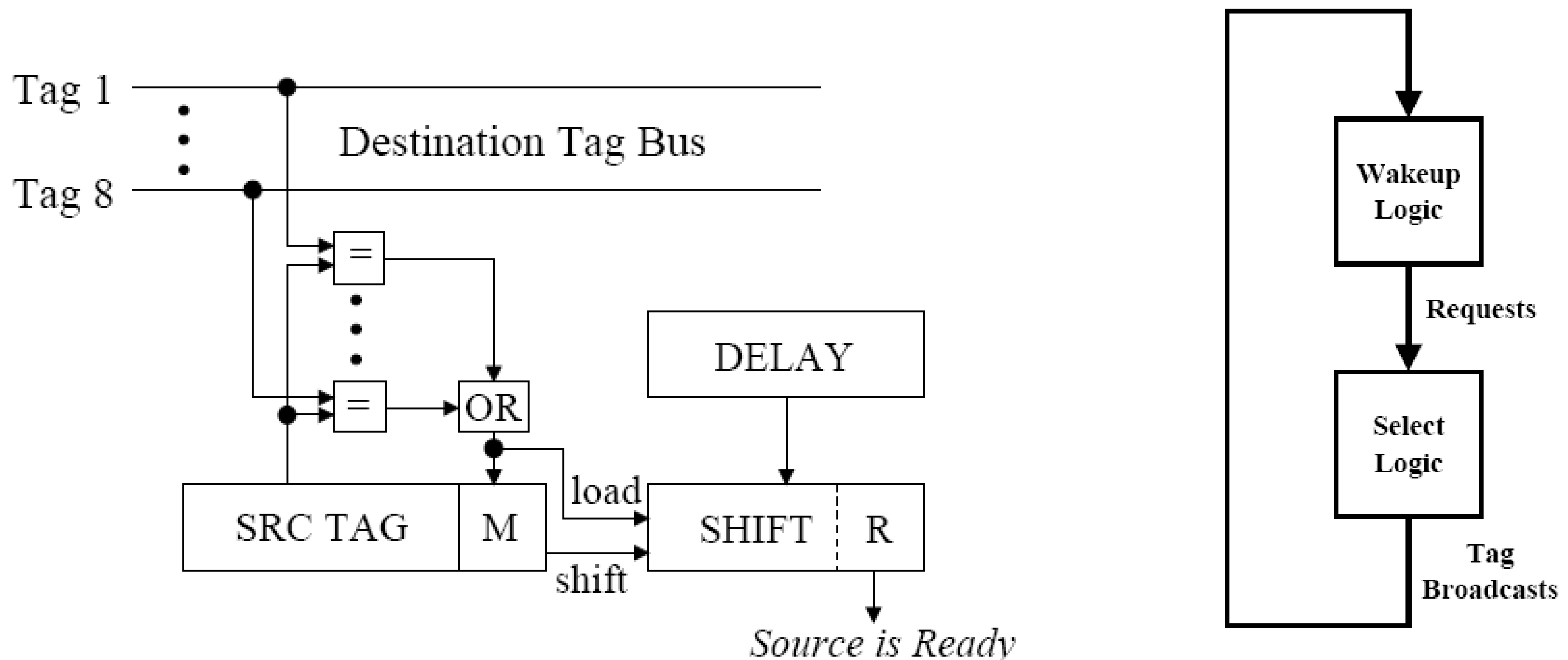
$$Delay = T_{tagdrive} + T_{tagmatch} + T_{matchOR}$$

$$T_{tagdrive} = c_0 + (c_1 + c_2 \times IW) \times WINSIZE + (c_3 + c_4 \times IW + c_5 \times IW^2) \times WINSIZE^2$$

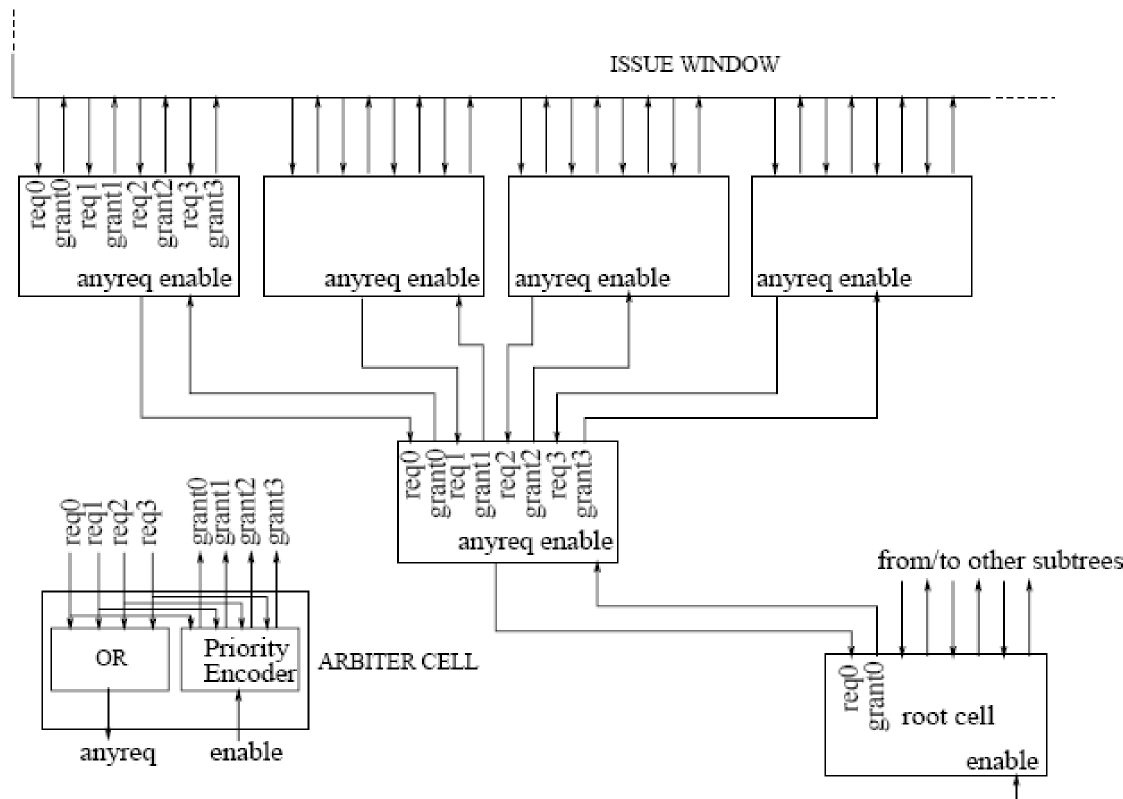
$$T_{tagmatch}, T_{matchOR} = c_0 + c_1 \times IW + c_2 \times IW^2$$

Wakeup Logic (II)

- Tag broadcast after selection
  - Sets countdown delay to the latency of selected instruction
  - When delay == 0, ready bit is set
  - Enables back-to-back operations without bubbles



Selection Logic



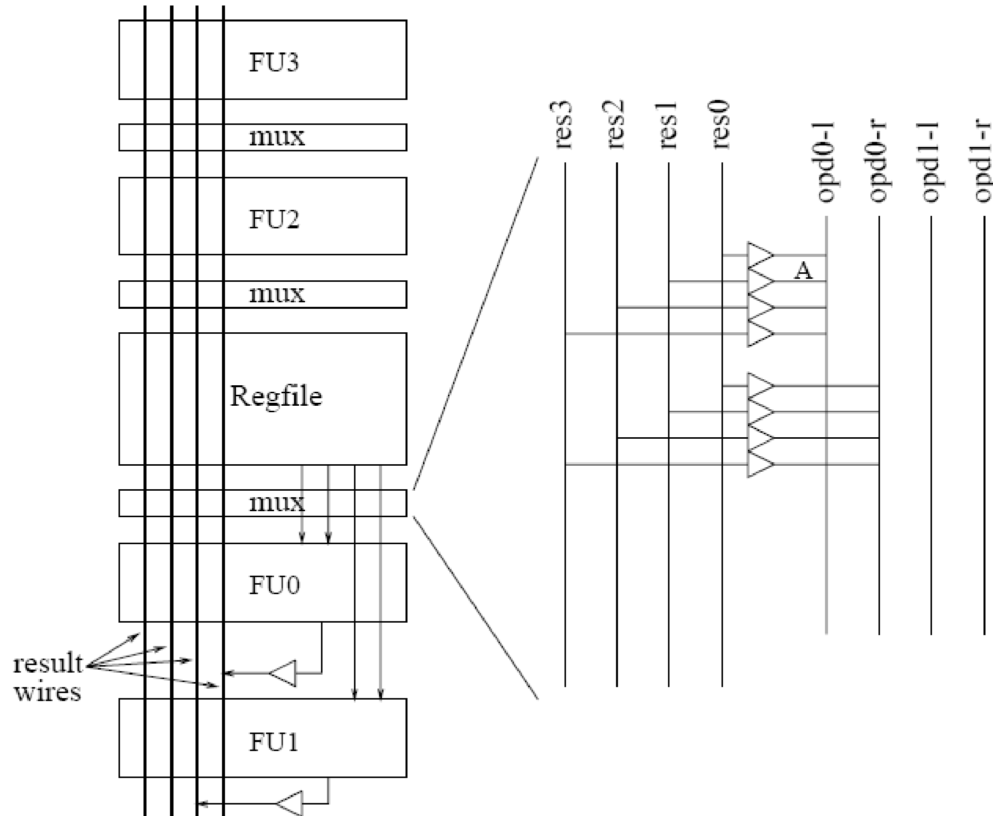
$$T_{selection} = c_0 + c_1 \times \boxed{log_4(WINSIZE)}$$

- Palacharla et al., "Complexity Effective Superscalar Processors," ISCA 1997.

Superscalar Processing

- Fetch (supply N instructions)
- Decode (generate control signals for N instructions)
- Rename (detect dependencies between N instructions)
- Dispatch (determine readiness and select N instructions to execute in-order or out-of-order)
- Execute (have enough functional units to execute N instructions + forwarding paths to forward results of N instructions)
- Write into Register File (have enough ports to write results of N instructions)
- Retire (N instructions)

Data Forwarding/Bypassing Between Multiple Functional Units



$$T_{bypass} = 0.5 \times R_{metal} \times C_{metal} \times L^2$$

- Broadcast results bypassed to all functional units
- Increasing issue width increases the length L of bypass wires (due to more functional units)
- Bypass delay scales quadratically with issue width

Empirical Delay Analysis

Issue width	Window size	Rename delay (ps)	Wakeup+Select delay (ps)	Bypass delay (ps)
0.8μm technology				
4	32	1577.9	2903.7	184.9
8	64	1710.5	3369.4	1056.4
0.35μm technology				
4	32	627.2	1248.4	184.9
8	64	726.6	1484.8	1056.4
0.18μm technology				
4	32	351.0	578.0	184.9
8	64	427.9	724.0	1056.4

- Palacharla et al., "Complexity Effective Superscalar Processors," ISCA 1997.

Reducing Dispatch+Bypass Delays

- Idea 1: Clustering (e.g., Alpha 21264 integer units)
  - Divide the scheduling window (and register file) into multiple clusters
  - Instructions steered into clusters (e.g. based on dependence)
  - Clusters schedule instructions out-of-order, within cluster scheduling can be in-order
  - Inter-cluster communication happens via register files (no full bypass)

  + Smaller scheduling windows, simpler wakeup algorithms

  + Smaller ports into register files

  + Faster within-cluster bypass

  -- Extra delay when instructions require across-cluster communication

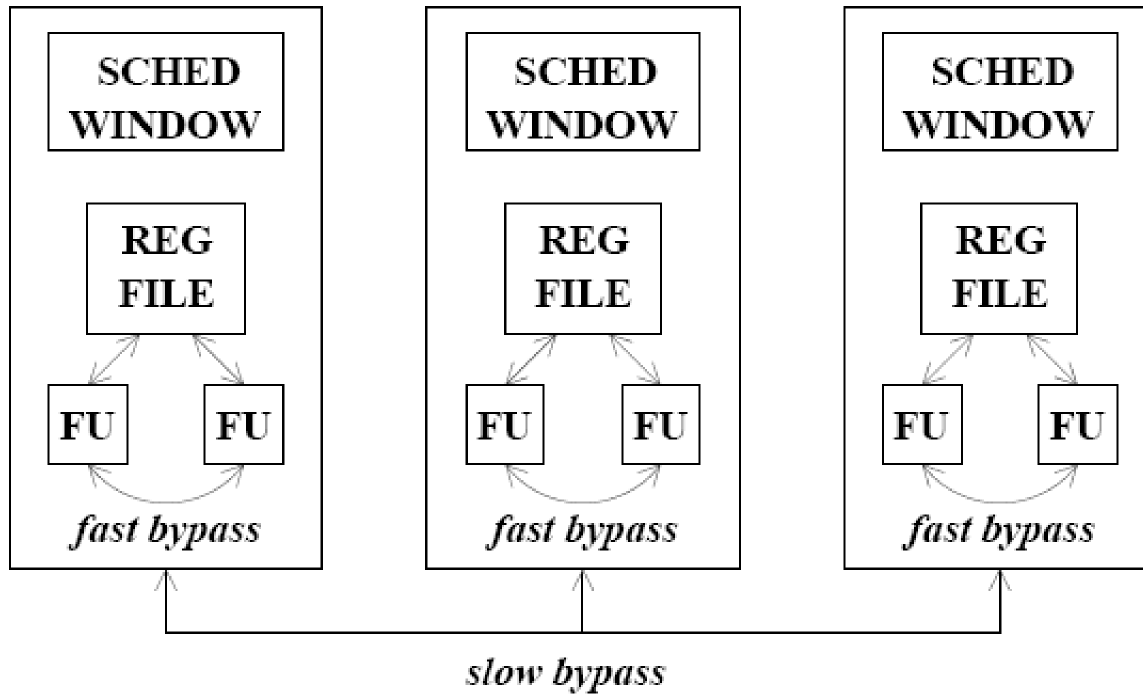
- Idea 2: Pipelining the Scheduling Logic (e.g., Pentium 4)

  + Breaks the wakeup + select loop

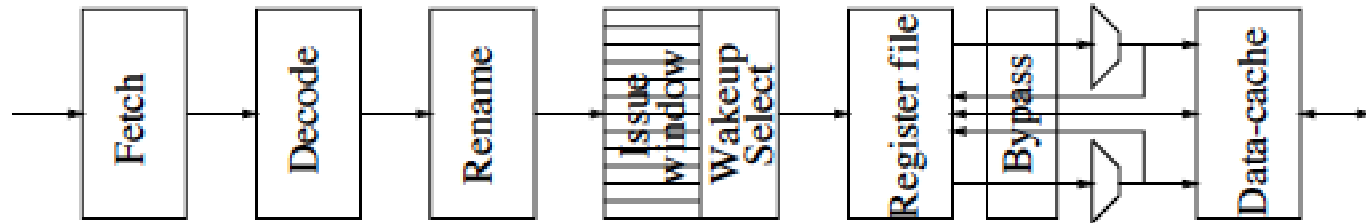
  -- Implementation complexity

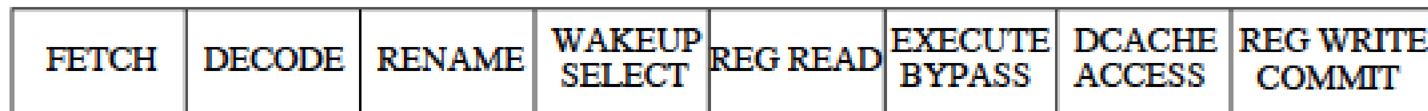
Clustering (I)

- Scheduling within each cluster can be out of order

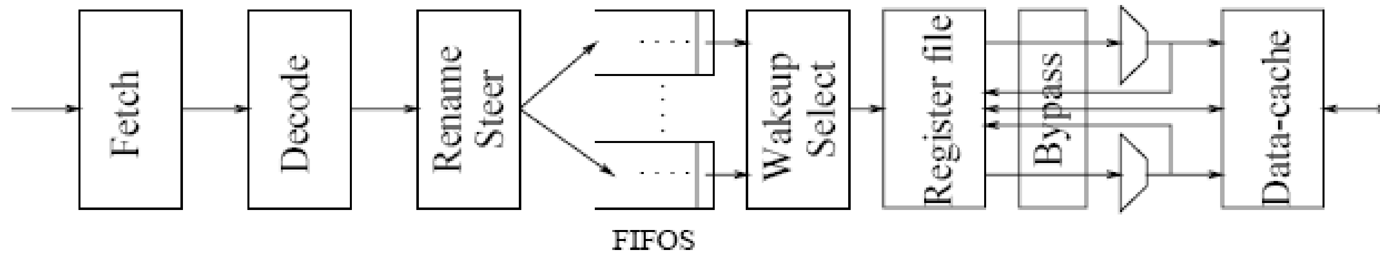


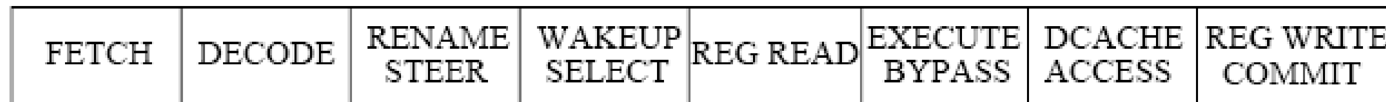
Clustering (II)





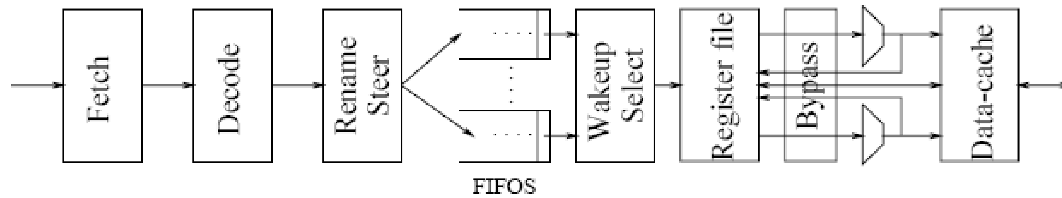
FETCH	DECODE	RENAME	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

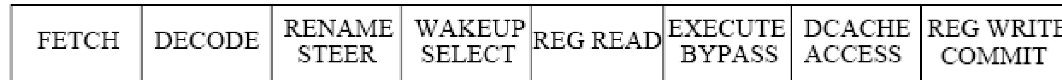


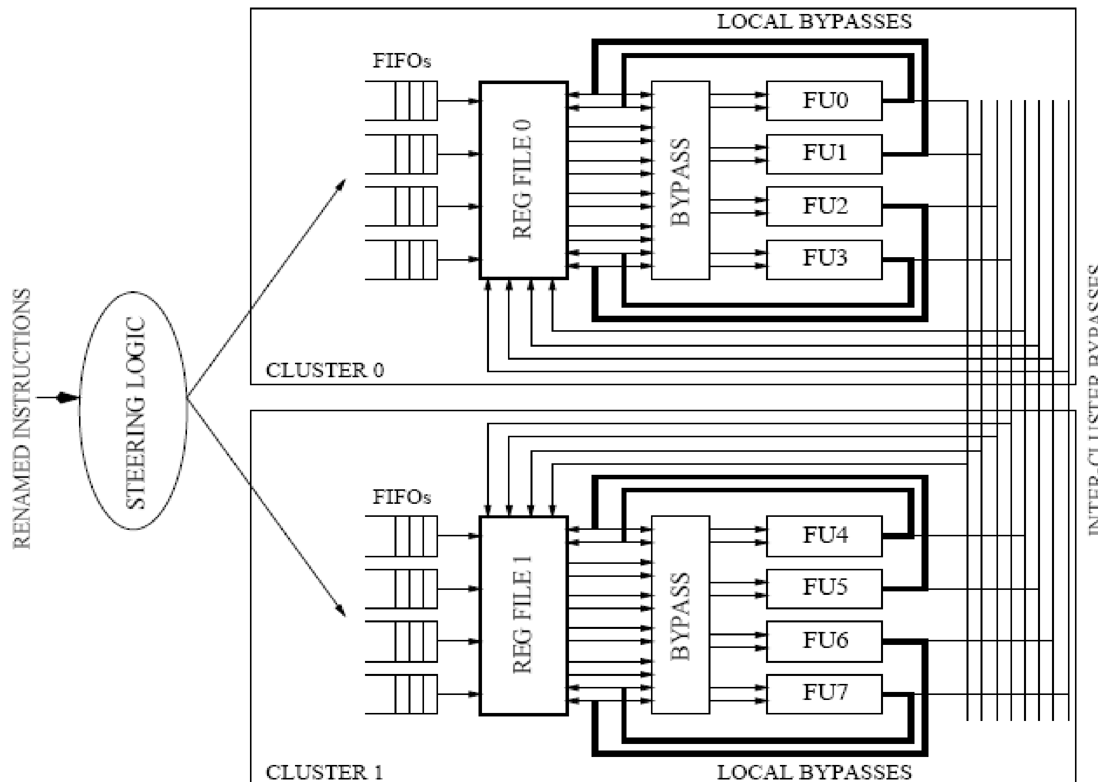


FETCH	DECODE	RENAME STEER	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

Clustering (III)







Each scheduler is a FIFO
+ Simpler
+ Can have N FIFOs
(OoO w.r.t. each other)
+ Reduces scheduling complexity
-- More dispatch stalls

Inter-cluster bypass: Results produced by an FU in Cluster 0 is not individually forwarded to each FU in another cluster.

- Palacharla et al., "Complexity Effective Superscalar Processors," ISCA 1997.

Superscalar Processing

- Fetch (supply N instructions)
- Decode (generate control signals for N instructions)
- Rename (detect dependencies between N instructions)
- Dispatch (determine readiness and select N instructions to execute in-order or out-of-order)
- Execute (have enough functional units to execute N instructions + forwarding paths to forward results of N instructions)
- Write into Register File (have enough ports to write results of N instructions)
- Retire (N instructions)

Multiple Instruction RegFile Read/Write

- Number of register file read/write ports scales linearly with the execution width
- Number of entries in the physical register file scales linearly with the instruction window size
- Increasing either increases RF access time

- Longer RF access → deeper pipeline

  -- Increased branch misprediction penalty

  -- Increased other misspeculation penalty (e.g. load store dependence misprediction)

Reducing RegFile Latency (I)

- Clustering
  - Register file can be partitioned or replicated across clusters
  - What if an instruction in one cluster needs a register in another?
    - Replicated: the register value produced in another cluster arrives N cycles later (Alpha 21264: 1 cycle inter-cluster delay)
      - Fewer read ports than monolithic register file
    - Partitioned: Special COPY instructions inserted to get the value from another cluster (to be stored in a buffer)
      - Fewer read and write ports than monolithic
    - Write specialization based clustering:
      - Instructions in a cluster can write to a subset of physical register file
      - Instructions can read from all subsets
      - M write ports per entry, N read ports (N > M)

Reducing RegFile Latency (II)

- Block Structured ISA
- Registers internal to a block do not need to be written to the external register file
  - Fewer write ports
  - Fewer external register file entries
- Registers internal to a block mostly communicated via forwarding paths or via an internal register file
  - Fewer read ports to the external register file

Superscalar Processing

- Fetch (supply N instructions)
- Decode (generate control signals for N instructions)
- Rename (detect dependencies between N instructions)
- Dispatch (determine readiness and select N instructions to execute in-order or out-of-order)
- Execute (have enough functional units to execute N instructions + forwarding paths to forward results of N instructions)
- Write into Register File (have enough ports to write results of N instructions)
- Retire (N instructions)

Multiple Instruction Retirement

- Retirement is not on the critical path
  - Increasing retirement pipeline depth does not affect misspeculation penalty
- Retirement functions
  - Check for exceptions
  - Update architectural state with instruction results
  - Deallocate pipeline resources allocated to instruction

- In many modern processors, architectural state update for a register-writing instruction is simply updating the retired register map (architectural register map) table to point to the destination physical register
  - Arch reg map points to arch state in the physical reg file

Deallocating Physical Registers (I)

- When is a physical register not needed any more? Two conditions:
  1. No instruction will source it
     i) No instruction in the pipeline needs it
     ii) Another instruction overwrote the same architectural register in the rename map
  2. Register value will not be needed to restore register file state due to an exception or a misprediction

- When an instruction updates the architectural register map for an architectural register:
  - The previous physical register mapped to the same arch reg can be deallocated

Deallocating Physical Registers (II)

- Can we deallocate a physical registers before their corresponding architectural register is written by a retired instruction?

- Idea: Early register recycling
  - Detect the two conditions under which the register can be deallocated → if satisfied, deallocate the register

+ Effectively reduces the physical register file size

-- Complexity of detecting conditions for recycling

Research Issues in Superscalar Processing (Not covered in lecture, just FYI)

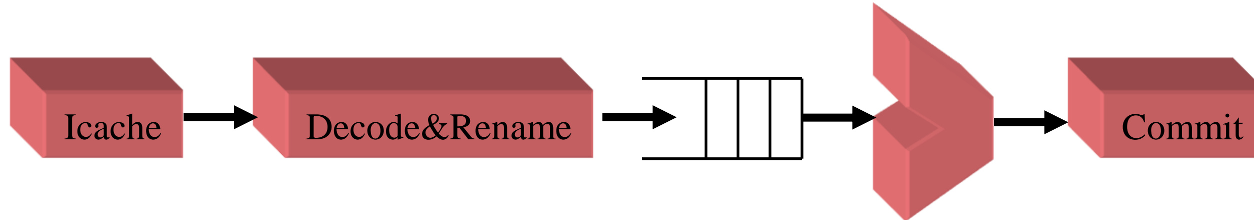
Research Issues in Superscalar Processing

- Simplifying the superscalar pipeline w/o losing performance
  - Clustering → how to steer instructions to clusters such that inter-cluster communication delay is avoided
  - More efficient utilization of on-chip resources
    - Register files, reservation stations
    - Recycle when (likely) not needed, virtualize as much as possible
  - Hardware/software co-designs
    - Software reduces the burden of hardware, e.g. BS-ISA
      - Wide fetch by code reordering
      - Partial renaming
      - Reducing register file size
      - Fetch → dispatch → FU instruction alignment
  - Atomic execution units can enable many simplifications
    - And, code optimizations
- Sharing superscalar resources between cores

Virtual-Physical Registers

- Monreal et al., "Delaying Physical Register Allocation through Virtual-Physical Registers," MICRO 1999.
  - Motivation: an instruction does not need the storage for its destination physical register until execution
  - Idea: When renaming, do not allocate a physical register, assign only a "virtual name"
    Upon dispatch (or end of execution), bind the virtual name to a physical register

  + More efficient utilization of physical register file space



Conventional renaming

Virtual-physical registers