CS 423
Computer Architecture
Spring 2012

Lecture 04:
A Superscalar Pipeline

Ozcan Ozturk

http://www.cs.bilkent.edu.tr/~ozturk/cs423/

[Adapted from *Computer Organization and Design*,
Patterson & Hennessy, © 2005, UCB]

Review: Pipeline Hazards

- Structural hazards
  - Design pipeline to eliminate structural hazards
- Data hazards – read before write
  - Use data forwarding inside the pipeline
  - For those cases that forwarding won't solve (e.g., load-use) include hazard hardware to insert stalls in the instruction stream
- Control hazards – `beq`, `bne`, `j`, `jr`, `jal`
  - Stall – hurts performance
  - Move decision point as early in the pipeline as possible – reduces number of stalls at the cost of additional hardware
  - Delay decision (requires compiler support) – not feasible for deeper pipes requiring more than one delay slot to be filled
  - Predict – with even more hardware, can reduce the impact of control hazard stalls even further if the branch prediction (BHT) is correct and if the branched-to instruction is cached (BTB)

Extracting Yet *More* Performance

- Two options:
  - Increase the depth of the pipeline to increase the clock rate – superpipelining (more details to come)
  - Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – multiple-issue

- Launching multiple instructions per stage allows the instruction execution rate, CPI, to be less than 1
  - So instead we use IPC: instructions per clock cycle
    - E.g., a 6 GHz, four-way multiple-issue processor can execute at a peak rate of 24 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4
  - If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?

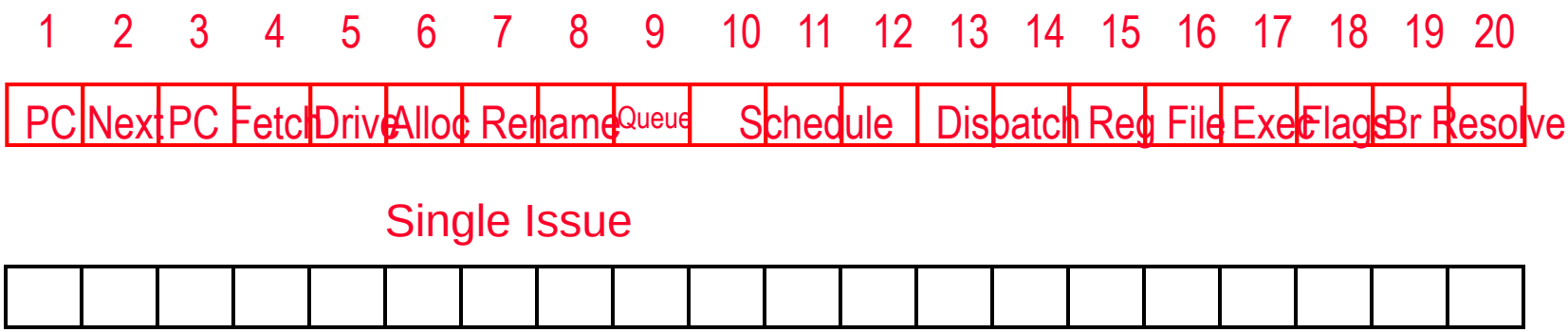
Superpipelined Processors

- Increase the depth of the pipeline leading to shorter clock cycles (and more instructions “in flight” at one time)
  - The higher the degree of superpipelining, the more forwarding/hazard hardware needed, the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time), and the bigger the clock skew issues (i.e., because of faster and faster clocks)

Superpipelined vs Superscalar

- Superpipelined processors have longer instruction latency than the SS processors which can degrade performance in the presence of true dependencies
- Superscalar processors are more susceptible to resource conflicts – but we can fix this with hardware !

Branch Misprediction



Instruction vs Machine Parallelism

- Instruction-level parallelism (ILP) of a program – a measure of the average number of instructions in a program that a processor *might* be able to execute at the same time
  - Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions
- Data-level parallelism (DLP)

```
DO  I = 1  TO  100
   A[I] = A[I] + 1
CONTINUE
```

- Machine parallelism of a processor – a measure of the ability of the processor to take advantage of the ILP of the program
  - Determined by the number of instructions that can be fetched and executed at the same time
- To achieve high performance, need *both* ILP and machine parallelism

Multiple-Issue Processor Styles

- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)

- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously are being made dynamically (at run time by the hardware)
  - E.g., IBM Power 2, Pentium 4, MIPS R10K, HP PA 8500

Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - Storage (data) dependencies – aka data hazards
    - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
  - Procedural dependencies – aka control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
  - Resource conflicts – aka structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

Instruction Issue and Completion Policies

- Instruction-issue – initiate execution
  - Instruction lookahead capability – fetch, decode and issue instructions beyond the current instruction
- Instruction-completion – complete execution
  - Processor lookahead capability – complete issued instructions beyond the current instruction
- Instruction-commit – write back results to the RegFile or D$ (i.e., change the machine state)

In-order issue with in-order completion

In-order issue with out-of-order completion

Out-of-order issue with out-of-order completion

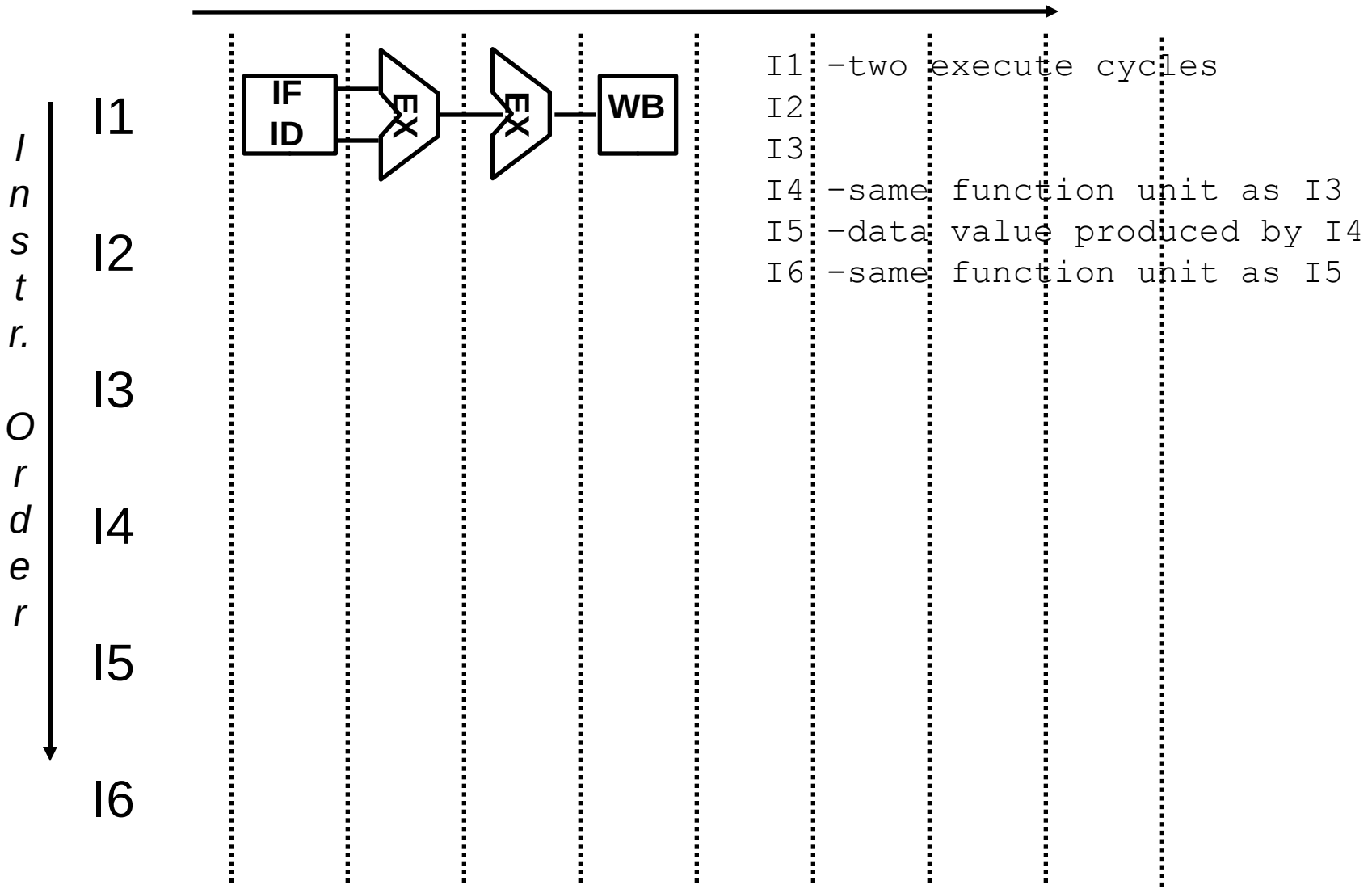
Out-of-order issue with out-of-order completion and in-order commit

In-Order Issue with In-Order Completion

- Simplest policy is to issue instructions in exact program order and to complete them in the same order they were fetched (i.e., in program order)
- Example:
  - Assume a pipelined processor that can fetch and decode two instructions per cycle, that has three functional units (a single cycle adder, a single cycle shifter, and a two cycle multiplier), and that can complete (and write back) two results per cycle
  - And an instruction sequence with the following characteristics

```
I1 – needs two execute cycles (a multiply)
I2
I3
I4 – needs the same function unit as I3
I5 – needs data value produced by I4
I6 – needs the same function unit as I5
```

In-Order Issue, In-Order Completion Example

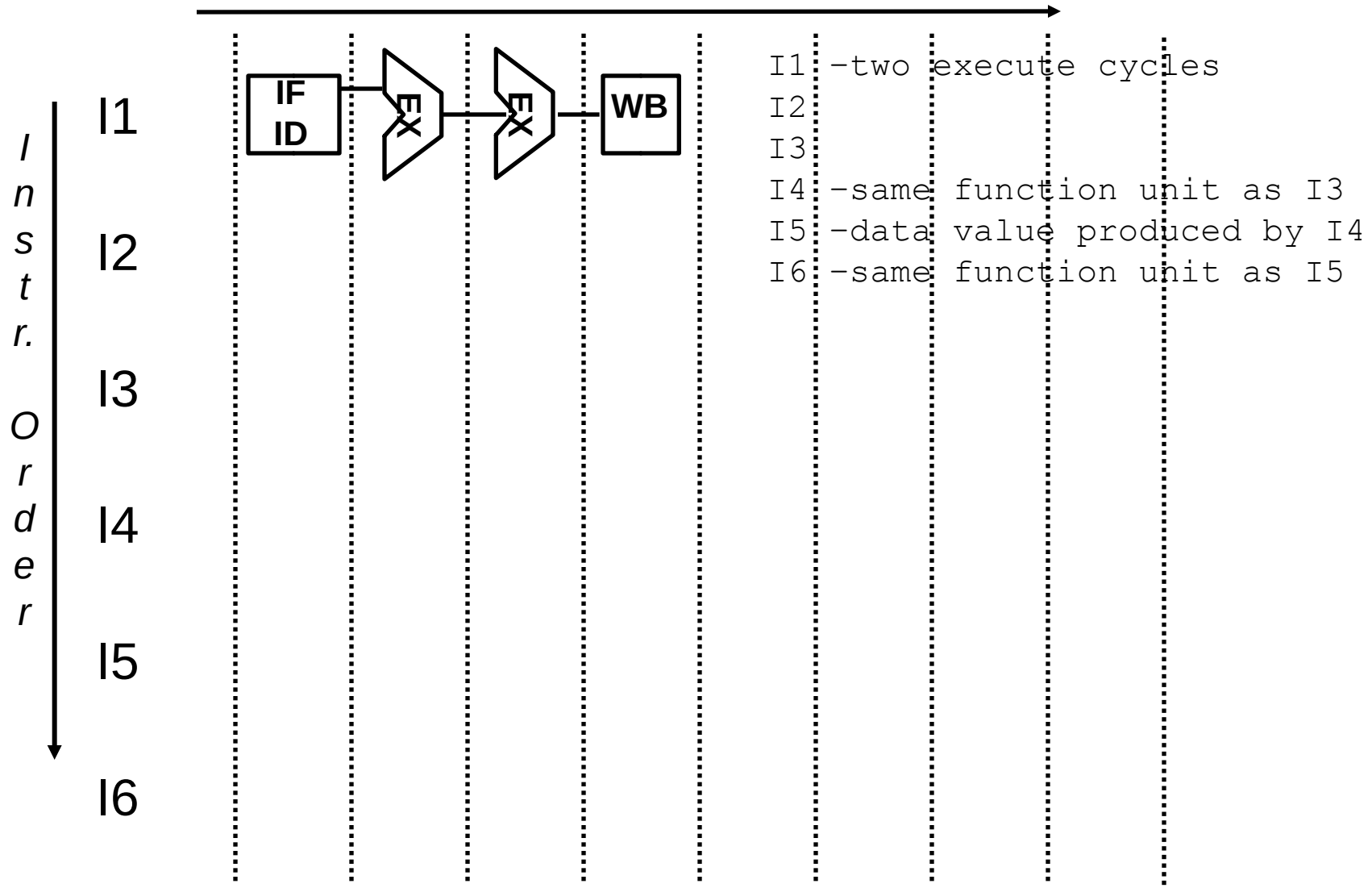


In-Order Issue with Out-of-Order Completion

- With out-of-order completion, a later instruction may complete before a previous instruction
  - Out-of-order completion is used in single-issue pipelined processors to improve the performance of long-latency operations such as divide

- When using out-of-order completion instruction issue is stalled when there is a resource conflict (e.g., for a functional unit) or when the instructions ready to issue need a result that has not yet been computed

IOI-OOC Example



Handling Output Dependencies

- There is one more situation that stalls instruction issuing with IOI-OOC, assume

  I1 – writes to R3
  I2 – writes to R3
  I5 – reads R3

  - If the I1 write occurs after the I2 write, then I5 reads an incorrect value for R3
  - I2 has an output dependency on I1 – write before write
    - The issuing of I2 would have to be stalled if its result might later be overwritten by an previous instruction (i.e., I1) that takes longer to complete – the stall happens before instruction issue

- While IOI-OOC yields higher performance, it requires more dependency checking hardware
  - Dependency checking needed to resolve both read before write *and* write before write

Out-of-Order Issue with Out-of-Order Completion

- With in-order issue the processor stops decoding instructions whenever a decoded instruction has a resource conflict or a data dependency on an issued, but uncompleted instruction
  - The processor is not able to *look beyond* the conflicted instruction even though more downstream instructions might have no conflicts and thus be issueable
- Fetch and decode instructions *beyond* the conflicted one, store them in an instruction buffer (as long as there's room), and flag those instructions in the buffer that don't have resource conflicts or data dependencies
- Flagged instructions are then issued from the buffer without regard to their program order

Antidependencies

- With OOI *also* have to deal with data antidependencies – when a later instruction (that completes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that issues later)

```
R3 := R3 * R5
R4 := R3 + 1
R3 := R5 + 1
```

True data dependency
Output dependency
Antidependency

- The constraint is similar to that of true data dependencies, except *reversed*
  - Instead of the later instruction using a value (not yet) produced by an earlier instruction (read before write), the later instruction produces a value that destroys a value that the earlier instruction (has not yet) used (write before read)

Dependencies Review

- Each of the three data dependencies
  - True data dependencies (read before write)
  - Antidependencies (write before read)
  - Output dependencies (write before write)

  } storage conflicts

  manifests itself through the use of registers (or other storage locations)

- True dependencies represent the flow of data and information through a program

- Anti- and output dependencies arise because the limited number of registers mean that programmers reuse registers for different computations

- When instructions are issued out-of-order, the correspondence between registers and values breaks down and the values *conflict* for registers

Storage Conflicts and Register Renaming

- Storage conflicts can be reduced (or eliminated) by increasing or duplicating the troublesome resource
  - Provide additional registers that are used to reestablish the correspondence between registers and values
    - Allocated dynamically by the hardware in SS processors
- Register renaming – the processor renames the original register identifier in the instruction to a new register (one not in the visible register set)

```
R3 := R3 * R5          R3b := R3a * R5a
R4 := R3 + 1     ⇨     R4a := R3b + 1
R3 := R5 + 1           R3c := R5a + 1
```

  - The hardware that does renaming assigns a “replacement” register from a pool of free registers and releases it back to the pool when its value is superseded and there are no outstanding references to it

Review: Extracting More Performance

- To achieve high performance, need both machine parallelism and instruction level parallelism (ILP) by
  - Superpipelining
  - Static multiple-issue (VLIW)
  - Dynamic multiple-issue (superscalar)
- A processor's instruction issue and completion policies impact available ILP
  - In-order issue with in-order completion
  - In-order issue with out-of-order completion
    - Creates output dependencies (write before write)
  - Out-of-order issue with out-of-order completion
    - Creates antidependency (write before read)
  - Out-of-order issue with out-of-order completion and in-order commit
- Register renaming can solve these storage dependencies

Speedup Measurements

- The speedup of the SS processor is
  - Assumes scalar and superscalar machines have the same IC & CR

$$\text{speedup} = s_n = \frac{\#\text{ scalar cycles}}{\#\text{ superscalar cycles}}$$

- To compute average speedup performance can use
  - Arithmetic mean

$$AM = 1/n \sum_{i=1}^{n} s_i$$

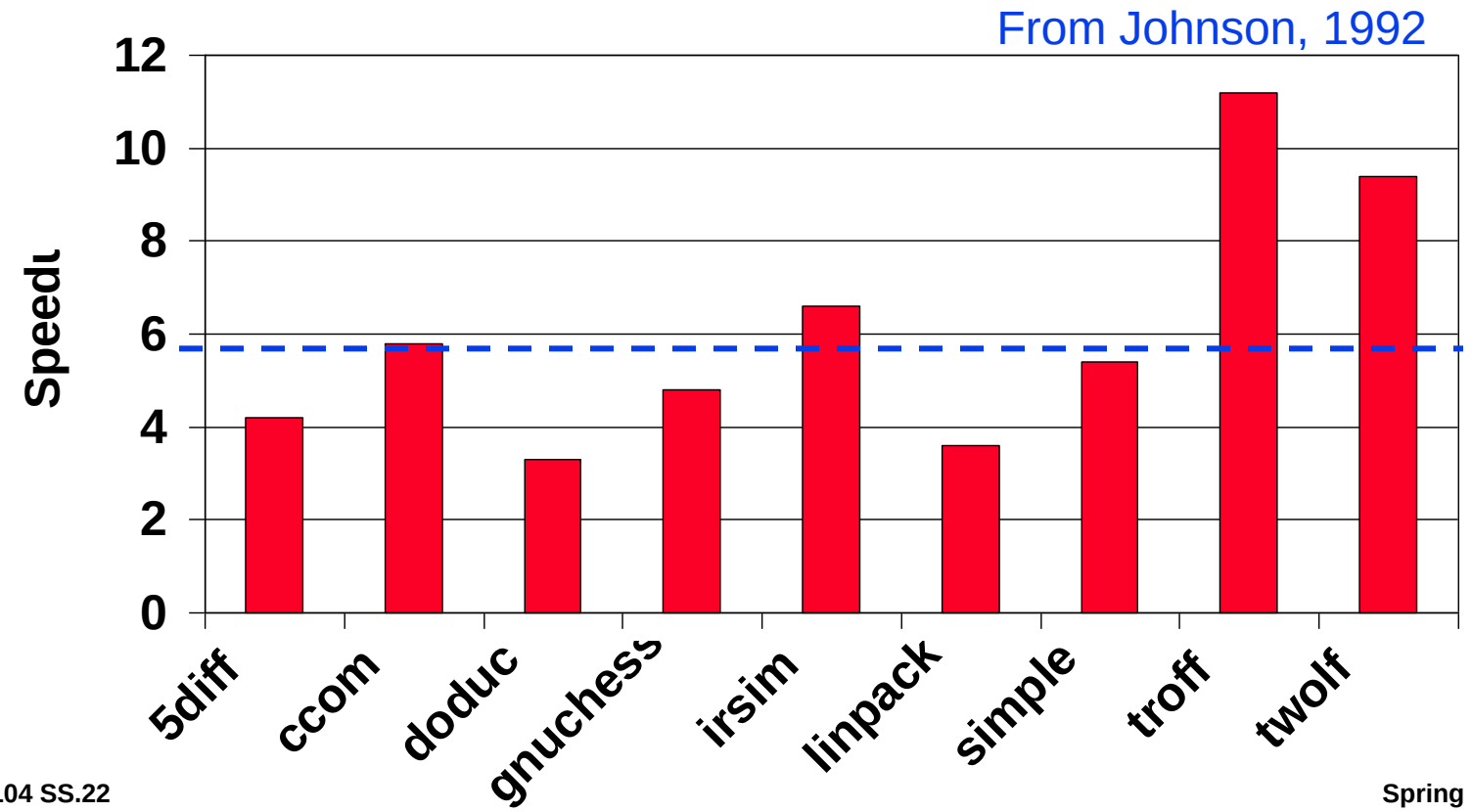
  - Harmonic mean

$$HM = n / \left(\sum_{i=1}^{n} 1/s_i \right)$$

    - assigns a larger weighting to the programs with the smallest speedup
  - EX: two programs with same scalar cycles, with a SS speedup of 2 for program1 and 25 for program2
    - AM = ½ * (2 + 25) = 13.5
    - HM = 2 / (.5 + .04) = 2 /.54 = 3.7

Maximum (Theoretical) SS Speedups

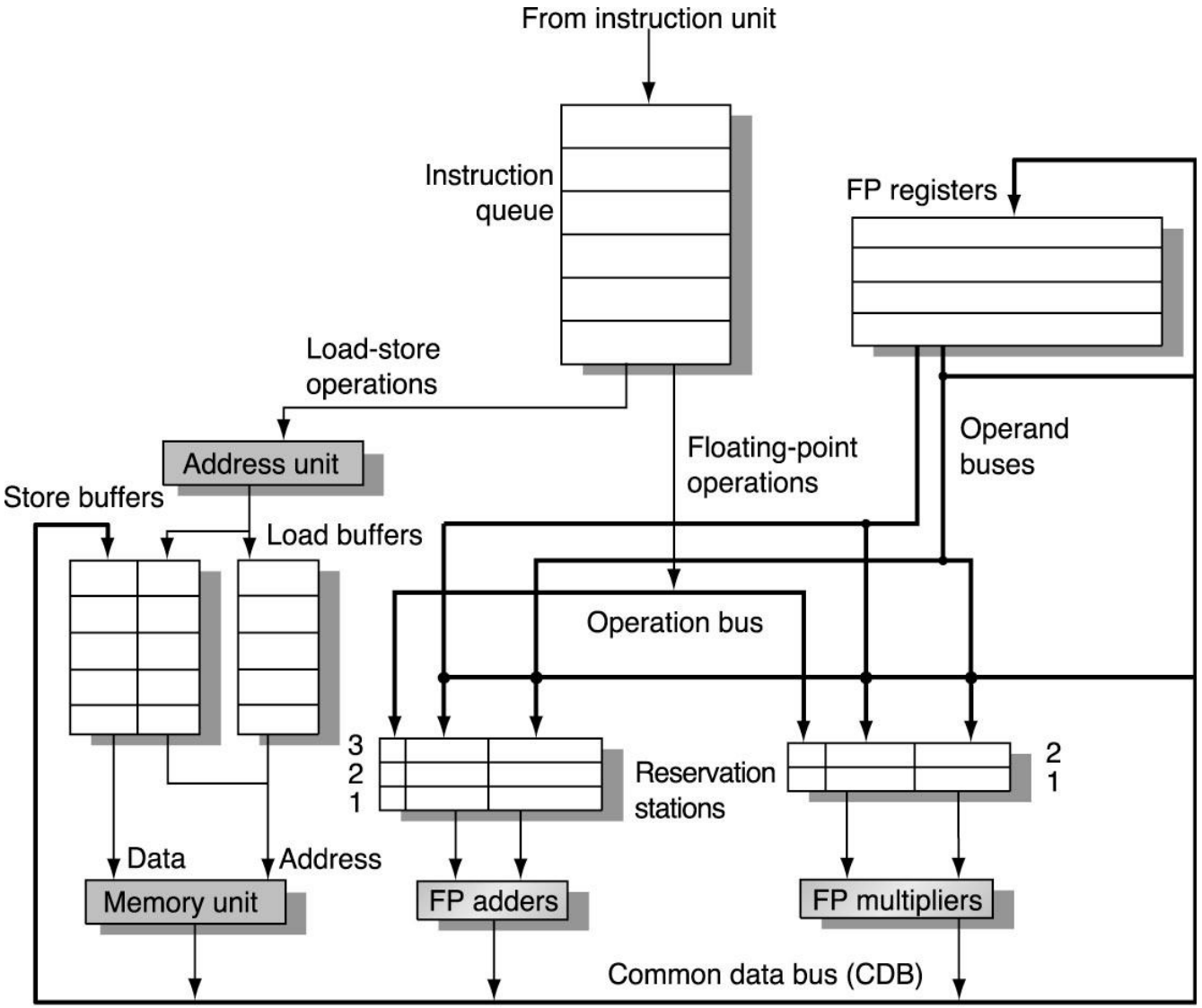
- The highest speedup that can be achieved with “ideal” machine parallelism (ignoring resource conflicts, storage dependencies, and procedural dependencies)
  - HM of 5.4 is the highest average speedup for these benchmarks that can be achieved even with ideal machine parallelism!



Tomasulo Algorithm

- For IBM 360/91 about 3 years after CDC 6600
- Goal: High Performance without special compilers
- Tomasulo Algorithm
  - Control & buffers distributed with Function Units called “reservation stations”
  - Registers in instructions replaced by pointers to reservation station buffer
  - HW renaming of registers to avoid WAW hazards
  - Buffer operand values to avoid WAR hazards
  - Common Data Bus broadcasts results to all FUs
  - Load and Stores treated as FUs as well
- Why study? Lead to Alpha 21264, HP 8000, MIPS 10000, Pentium II, Power PC 604 …

FP unit and load-store unit using Tomasulo's alg.



Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

   Stall if structural hazard, ie. no space in the rs. If reservation station (*rs*) is free, the issue logic issues instr to *rs* & read operands into *rs if ready (Register renaming => Solves WAR). Make status of destination register waiting for this latest instn even if the previous instn writing to this register hasn't completed => Solves WAW hazards*.

2. Execution—operate on operands (EX)

   When both operands are ready then execute;
   if not ready, watch CDB for result – Solves RAW

3. Write result—finish execution (WB)

   Write on Common Data Bus to all awaiting units;
   mark reservation station available. Write result into dest. reg. if its status is *r*. => Solves WAW.

- CDB: data + source ("come from" bus)
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does broadcast

Reservation Station Components

Op—Operation to perform in the unit (e.g., + or –)

Vj, Vk— Value of the source operand.

Qj, Qk— Name of the RS that would provide the source operands. Value zero means the source operands already available in Vj or Vk, or is not necessary.

Busy—Indicates reservation station or FU is busy

Register File Status Qi:

Qi —Indicates which functional unit will write each register, if one exists. Blank (0) when no pending instructions that will write that register meaning that the value is already available.

Tomasulo Example Cycle 0

Instruction status					Execution	Write						
Instruction		j	k	Issue	complete	Result			Busy	Address		
LD	F6	34+	R2					Load1	No			
LD	F2	45+	R3					Load2	No			
MULTD	F0	F2	F4					Load3	No			
SUBD	F8	F6	F2									
DIVD	F10	F0	F6									
ADDD	F6	F8	F2									
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	No									
	0	Mult2	No									
Register result status												
Clock				F0	F2	F4	F6	F8	F10	F12	...	F30
0			FU									

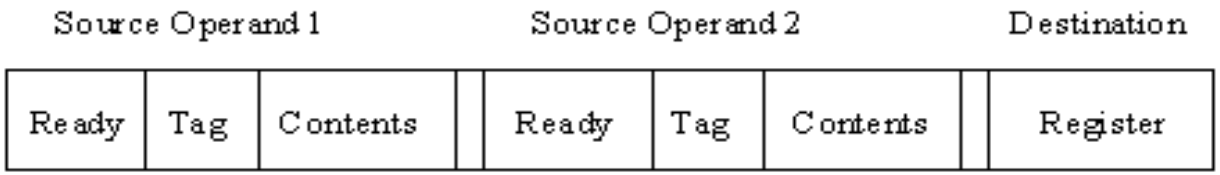
Tomasulo Drawbacks

- Complexity
- Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
  - Each CDB must go to multiple functional units
    $\Rightarrow$high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one!
    - Multiple CDBs $\Rightarrow$ more FU logic for parallel assoc stores

Dependency Resolution

Introduction to Tomasulo's Dependency-resolution Algorithm

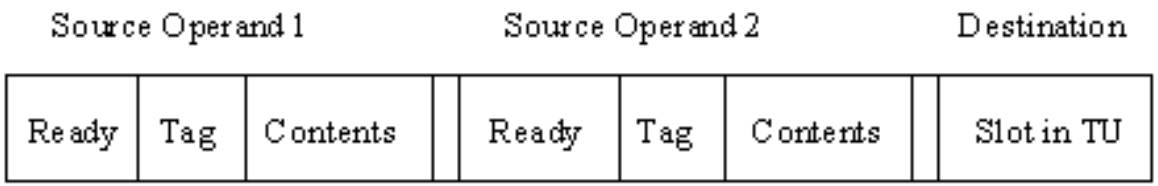
- When an instruction enters the decode and issue stage and its operands are not available, it is forwarded to a *Reservation Station (RS)* associated with the functional unit that it will be using.
  - If a source register is busy, the tag for the source register is obtained.
  - If the sink register (destination register) is busy, the instruction fetches a new tag, updates the tag of the sink register and proceeds to a RS.
- It waits in the RS until its data dependencies have been resolved.
- Once at a reservation station, an instruction can resolve its dependencies by monitoring the Common Data Bus (the Result Bus). When all the operands for an instruction are available, it is dispatched to the appropriate functional unit for execution.
- *Problem: Each register needs a tag. Associative comparison is needed.*



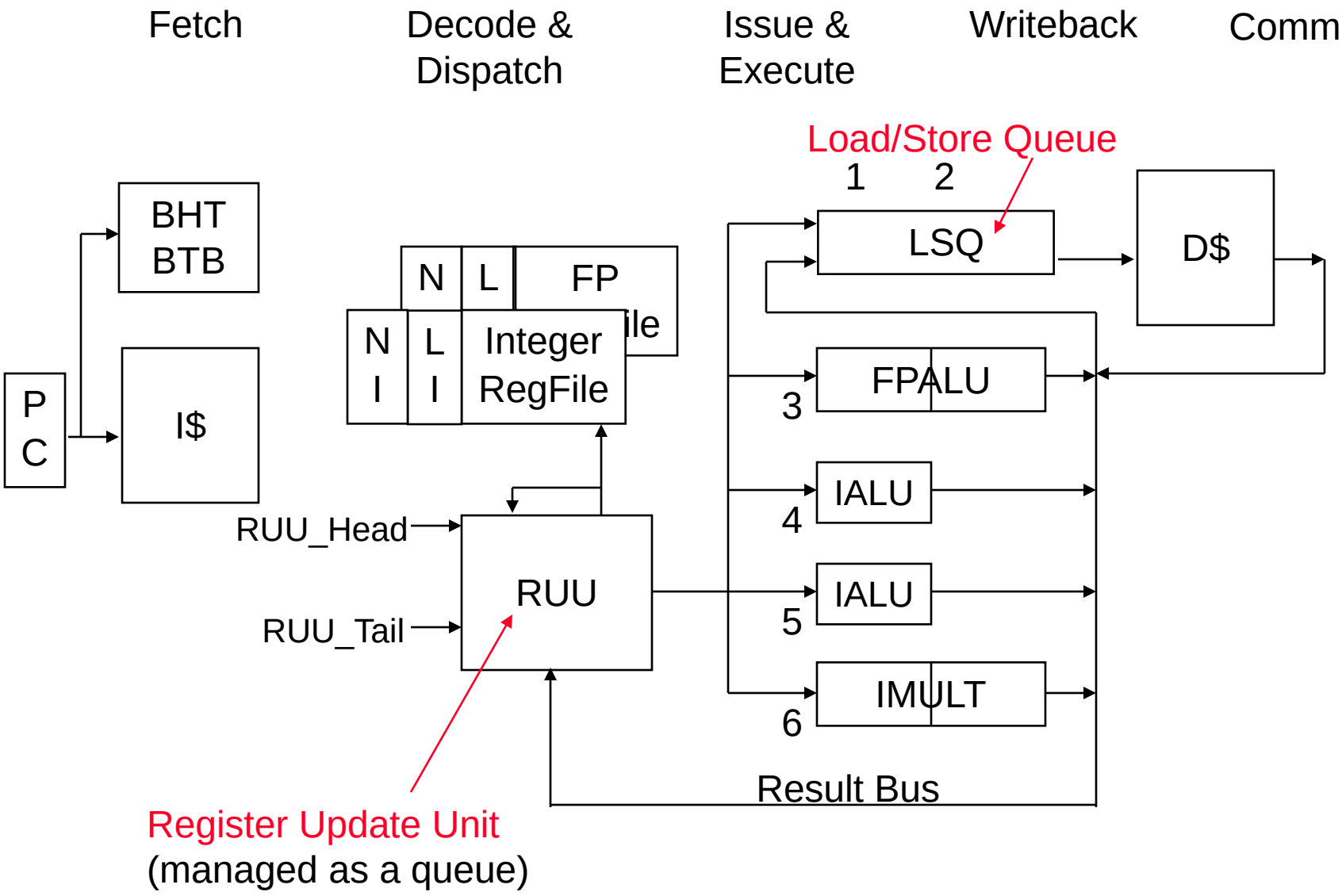
Dependency Resolution

Extensions to Tomasulo's Algorithm

- **Major improvement:** Consolidate the tags from all currently active registers into **one Tag Unit**. Each register has only a single busy bit.
- If a source register is busy, the current tag is got from the TU.
- For the destination register, a new tag is obtained.
  - If the destination register is not busy, it is easy to just get a new tag
  - If the destination register is busy, a new tag is obtained and the instruction holding the old tag is informed.
- The result from a functional unit (along with its tag) is broadcast to all reservation stations and is also forwarded to the TU. The TU then forwards the result to the appropriate register.



Baseline Superscalar MIPS Processor Model



Typical Functional Unit Latencies

- Result latency – number of cycles taken by a functional unit (FU) to produce a result
- Issue latency – minimum number of cycles between the issuing of an instruction to a FU and the issuing of the next instruction to the same FU

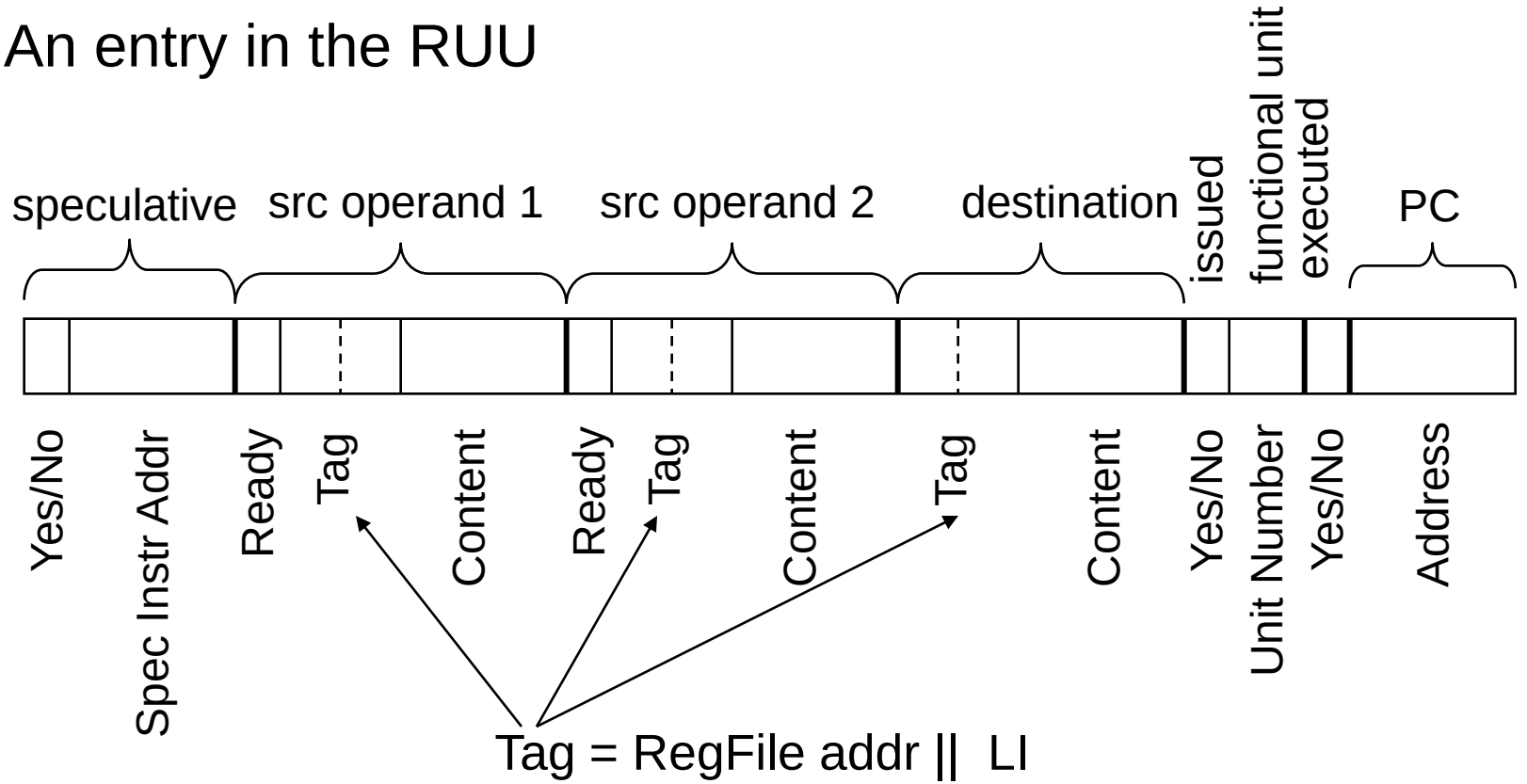
	Issue Latency	Result Latency
Integer ALU	1	1
Integer multiply	1	2
Load (on hit)	1	1
Load (on miss)	1	40
Store	1	n/a
FltPt Add	1	2
FltPt Multiply	1	4
FltPt Divide	12	12
FltPt Convert	1	2

Additional RegFile Fields

- Each register in the general purpose RegFile has two associated n-bit counters (n of 3 is typical)
  - NI (number of instances) – the number of instances of a register as a destination register in the RUU
  - LI (latest instance) – the number of the latest instance
- When an instruction with destination register address Ri is dispatched to the RUU, both its NI and LI are incremented
  - Dispatch is blocked if a destination register's NI is 2^n -1, so only up to 2^n – 1 instances of a register can be present in the RUU at any one time
- When an instruction is committed (updates the Ri value) the associated NI is decremented
  - When NI = 0 the register is "free" (there are no instruction in the RUU that are going to write to that register) and LI is cleared

Register Update Unit (RUU)

- A hardware data structure that is used to resolve data dependencies by keeping track of an instruction's data and execution needs and that commits completed instructions in program order
- An entry in the RUU



Basic Instruction Flow Overview

- Fetch (in program order): Fetch multiple instructions in parallel from the I$
- Decode & Dispatch (in program order):
  - In parallel, decode the instr's just fetched and schedule them for execution by dispatching them to the RUU
  - Loads and stores are dispatched as two (micro)instr's – one to compute the effective addr and one to do the memory operation
- Issue & Execute (out of program order): As soon as the RUU has the instr's source data and the FU is free, the instr's are issued to the FU for execution
- Writeback (out of program order): When done the FU puts its results on the Result Bus which allows the RUU and the LSQ to be updated – the instr completes
- Commit (in program order): When appropriate, commit the instr's result data to the state locations (i.e., update D$ and RegFile)

Managing the RUU as a Queue

- By managing the RUU as a queue, and committing instruction from RUU_Head, instruction are committed (aka retired) in the order they were received from the Decode & Dispatch logic (in program order)
  - Stores to state locations (RegFile and D$) are buffered (in the RUU and LSQ) until commit time
  - Supports precise interrupts (the *only* state locations updated are those written by instructions *before* the interrupting instr)
- The counter (LI) allows multiple instances of a specific destination register to exist in the RUU at the *same* time via register renaming
  - Solves write before write hazards if results from the RUU are returned to the RegFile in program order
- Managing the RUU as a queue and committing from the head of the queue provides just this!

Major Functions of the RUU

- Each of the tasks are done in parallel every cycle

1. Accepts new instructions from the Decode & Dispatch logic
2. Monitors the Result Bus to resolve true dependencies and to do write back of result data to the RUU
3. Determines which instructions are ready for execution, reserves the Result Bus, and issues the instruction to the appropriate FU
4. Determines if an instruction can commit (i.e., change the machine state) and commits the instruction if appropriate

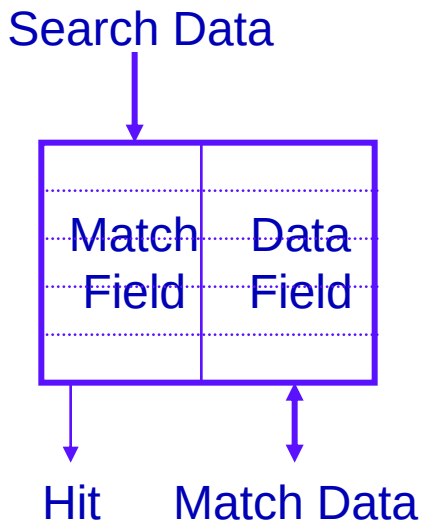
The First Function of the RUU

1. Accepts new instructions from the Decode & Dispatch logic – for each instruction in the fetch packet
   - The dispatch logic gets an entry in the RUU (a circular queue)
     - The RUU_Tail entry (currently empty) is allocated to the instruction and RUU_Tail is updated
     - Then if RUU_Head = RUU_Tail, the RUU is full and further instruction fetch stalls until the RUU_Head advances (as a result of a commit)
   - For each source operand, if the contents of the source register is available, then it is copied to the source Content field of the RUU entry and its Ready bit is set. If not, the source RegFile addr || LI is copied to the source Tag field and the Ready bit is reset.
   - For the destination operand, the RegFile destination addr || LI is copied to the allocated RUU destination Tag field
   - The issued bit and executed bit are set to No, the number of the FU needed for the operation is entered, and the PC address of the instruction is copied to the PC Address field

Aside: Content Addressable Memories (CAMs)

- Memories that are addressed by their content. Typical applications include RUU source tag field comparison logic, cache tags, and translation lookaside buffers

  - Memory hardware that compares the Search Data to the Match Field entries for *each* word in the CAM in *parallel* !
  - On a match the Data Field for that entry is output to Match Data on read or Match Data is written into the Data Field on write and the Hit bit is set.
  - If no match occurs, the Hit bit is reset.
  - CAMs can be designed to accommodate multiple hits.



The Second Function of the RUU

2. Monitors the Result Bus to resolve true dependencies and to do write back of result data to the RUU
   - The Result Bus destination addr || LI is compared associatively to the source Tag fields (for those source operands that are not Ready). If a match occurs, the data on the Result Bus is gated into the Content field for matching source operands.
     - The Result Bus contains the result data, its RUU entry address, and its RegFile destination addr || LI
   - The result data is gated into the destination Content field of the RUU entry that matches the RUU entry addr on the Result Bus
   - The executed bit is set to Yes

- Resolves true dependencies through the Ready bit (i.e., must wait for the source operands before issue)
- Solves anti-dependencies through LI (making sure that the source fields get updated only for the *correct* version of the data)

The Third Function of the RUU

3. Determines which instructions are ready for execution, reserves the Result Bus, and issues the ready instructions to the appropriate FU's for execution
   - When both source operands of an RUU entry are Ready, the RUU issues the highest priority instruction – priority is given to load and store instr's and then to the instr's that entered the RUU the earliest (i.e., the ones closest to RUU_Head).
   - Reserves the Result Bus
   - Issues the instruction (sends to the FU the source operands, RUU entry addr, and the destination's RegFile addr || LI) and sets the issued bit to Yes

- Multiple instructions can be issued in parallel, if they are ready, if they can reserve the Result Bus, and if they are destined for different FU's

The Fourth Function of the RUU

4. Determines if an instruction can commit (i.e., change the machine state) and commits the instruction if appropriate
   - Monitors the executed bit of the RUU_Head entry. If the bit is set, the destination content data is written into the RegFile at the destination's RegFile address
   - Matches the destination RegFile addr || LI against the RUU's source Tag fields and on match copies the destination content data into source Content fields
   - Decrements the associated RegFile entry's NI counter
   - Releases the RUU entry by incrementing the RUU_Head pointer

- Solves output dependencies by writing to RegFile in program order
- Multiple instructions can commit in parallel if they are ready to commit, if they are writing to different RegFile registers, and if there are multiple RegFile write ports

MicroOperations of Load and Store

- Recall that loads and stores are dispatched to the RUU as two (micro)instr's – one to compute the effective addr and one to do the memory operation
  - Load `lw R1,2(R2)` becomes
    ```
    addi    R0,R2,2
    lw      R1,R0
    ```
  - Store `sw R1,6(R2)` becomes
    ```
    addi    R0,R2,6
    sw      R1,R0
    ```
- At the same time a LSQ entry is allocated
  - Each LSQ entry consists of a Tag field (RegFile addr || LI) and a Content field. The LI counter allows for multiple instances of stores (writes) to a memory address
  - When a load completes (the D$ returns the data on the Result Bus) or a store commits (in program order) the LSQ entry is released
  - Instruction dispatch is blocked if there is not a free LSQ entry and two free RUU entries

Loads from Memory

- When a load's address becomes known, the address is compared (associatively) to see if it matches an entry already in the LSQ (i.e., if there is a pending operation to the same memory address)
  - If the match in the LSQ is for a load, the current load does not need to be issued (or executed) since the matching pending load will load in the data
  - If the match in the LSQ is for a store, the current load does not need to be issued (or executed) since the matching pending store can directly supply the destination Content for the current load
- If there is no match, the load is issued to the LSQ and executed when the D$ is next available
- When the RUU# of the load instr appears on the Result Bus (along with the memory data), the load completes by updating the RUU and releasing the LSQ entry (the RUU entry is released on load commit)

Stores to Memory

- When a store's address (and the store data) becomes known, the address is compared (associatively) to see if it matches an entry already in the LSQ (i.e., if there is a pending operation to the same memory address)
  - If the match in the LSQ is for a load, the current store is issued to the LSQ
  - If the match in the LSQ is for a store, the current store is issued to the LSQ with an incremented LI
  - If there is no match, the store is dispatched to the LSQ

- Stores are held in the LSQ until the store is ready to commit (i.e., until its partner instr reaches the RUU_Head) at which time the store is executed (i.e., the data and address are sent to the D$) and the RUU and LSQ entries are released

SimpleScalar Structure

- `sim-outorder`: supports out-of-order issue and execution (with in-order commit) with a Register Update Unit (RUU)
  - Uses a RUU for register renaming and to hold the results of pending instructions. The RUU (aka reorder buffer (ROB)) retires completed instructions in program order to the RegFile
  - Uses a LSQ for store instructions not ready to commit and load instructions waiting for access to the D$
  - Loads are satisfied by either the memory or by an earlier store value residing in the LSQ if their addresses match
    - Loads are issued to the memory system only when addresses of *all* previous loads and stores are known

Simulated SimpleScalar Pipeline

- `ruu_fetch()`: fetches instr's from one I$ line, puts them in the fetch queue, probes the cache line predictor to determine the next I$ line to access in the next cycle
  - fetch:ifqsize<size>: fetch width (default is 4)
  - fetch:speed<ratio>: ratio of the front end speed to the execution core (<ratio> times as many instructions fetched as decoded per cycle)
  - fetch:mplat<cycles>: branch misprediction latency (default is 3)
- `ruu_dispatch()`: decodes instr's in the fetch queue, puts them in the dispatch (scheduler) queue, enters and links instr's into the RUU and the LSQ, splits memory access instructions into two separate instr's (one to compute the effective addr and one to access the memory), notes branch mispredictions
  - decode:width<insts>: decode width (default is 4)

SimpleScalar Pipeline, con't

- `ruu_issue()` and `lsq_refresh()`: locates and marks the instr's ready to be issued by tracking register and memory dependencies, ready loads issued to D$ unless there are earlier stores in LSQ with unresolved addr's, forwards store values with matching addr to ready loads
  - issue:width<insts>: maximum issue width (default is 4)
  - ruu:size<insts>: RUU capacity in instr's (default is 16, min is 2)
  - lsq:size<insts>: LSQ capacity in instr's (default is 8, min is 2)

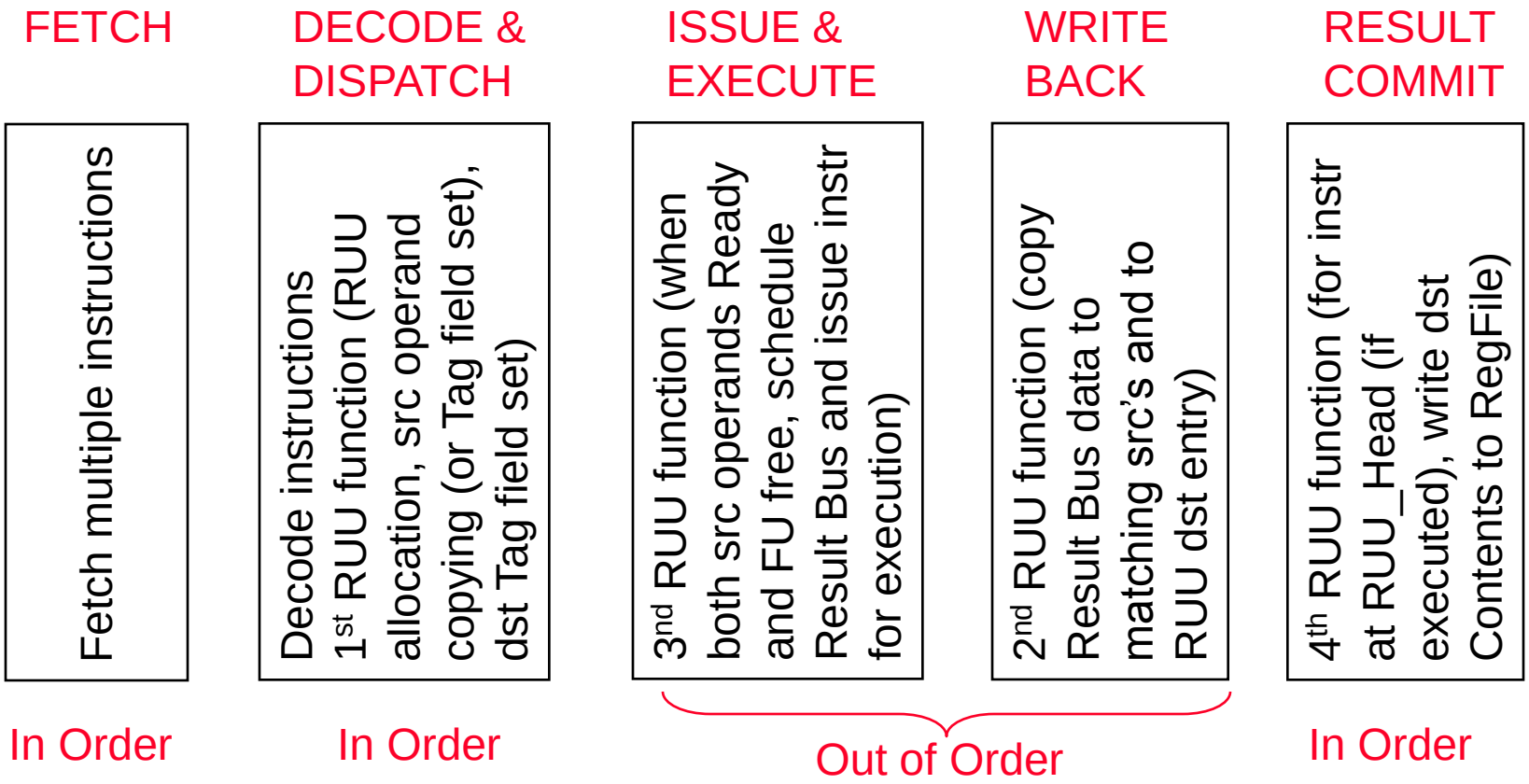
  and handles instr's execution – collects all the ready instr's from the scheduler queue (up to the issue width), check on FU availability, checks on access port availability, schedules writeback events based on FU latency (hardcoded in `fu_config[]`)
  - res:ialu | imult | memport | fpalu | fpmult<num>: number of FU's (default is 4 | 1 | 2 | 4 | 1)

SimpleScalar Pipeline, con't

- `ruu_writeback()`: determines completed instr's, does data forwarding to dependent waiting instr's, detects branch misprediction and on misprediction rolls the machine state back to the checkpoint and discards erroneously issued instructions

- `ruu_commit()`: in-order commits results for instr's (values copied from RUU to RegFile or LSQ to D$), RUU/LSQ entries for committed instr's freed; keeps retiring instructions at the head of RUU that are ready to commit until the head instr is one that is not ready

SS Pipeline


FETCH
DECODE & DISPATCH
ISSUE & EXECUTE
WRITE BACK
RESULT COMMIT
Fetch multiple instructions
Decode instructions
1st RUU function (RUU allocation, src operand copying (or Tag field set), dst Tag field set)
3nd RUU function (when both src operands Ready and FU free, schedule Result Bus and issue instr for execution)
2nd RUU function (copy Result Bus data to matching src's and to RUU dst entry)
4th RUU function (for instr at RUU_Head (if executed), write dst Contents to RegFile)
In Order
In Order
Out of Order
In Order

Our SS Model Performance

- Out of order issue has consistently the best performance for the benchmark programs

From Johnson, 1992