

# ECE/CS 757: Advanced Computer Architecture II

Instructor: Mikko H Lipasti

Spring 2009  
University of Wisconsin-Madison

Lecture notes based on slides created by John Shen, Mark Hill, David Wood, Guri Sohi, and Jim Smith, Natalie Enright Jerger, and probably others

## Review of 752

- Iron law
- Beyond pipelining
- Superscalar challenges
  - Instruction flow
  - Register data flow
  - Memory Dataflow
- Modern memory interface

## Iron Law

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$
$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size)                      (CPI)                      (cycle time)

Architecture --> Implementation --> Realization  
Compiler Designer    Processor Designer    Chip Designer

## Iron Law

- Instructions/Program
  - Instructions executed, not static code size
  - Determined by algorithm, compiler, ISA
- Cycles/Instruction
  - Determined by ISA and CPU organization
  - Overlap among instructions reduces this term
- Time/cycle
  - Determined by technology, organization, clever circuit design

## Our Goal

- Minimize time, which is the product, NOT isolated terms
- Common error to miss terms while devising optimizations
  - E.g. ISA change to decrease instruction count
  - BUT leads to CPU organization which makes clock slower
- Bottom line: terms are inter-related

## Pipelined Design

- Motivation:
    - Increase throughput with little increase in hardware.
- Bandwidth or Throughput = Performance
- Bandwidth (BW) = no. of tasks/unit time
  - For a system that operates on one task at a time:
    - BW = 1/delay (latency)
  - BW can be increased by pipelining if many operands exist which need the same operation, i.e. many repetitions of the same task are to be performed.
  - Latency required for each task remains the same or may even increase slightly.

### Ideal Pipelining

- Bandwidth increases linearly with pipeline depth
- Latency increases by latch delays

### Example: Integer Multiplier

- 16x16 combinational multiplier
- ISCAS-85 C6288 standard benchmark
- Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC

[Source: J. Hayes, Univ. of Toronto]

### Example: Integer Multiplier

Configuration	Delay	MPS	Area (FF/wiring)	Area Increase
Combinational	3.52ns	284	7535 (-/1759)	
2 Stages	1.87ns	534 (1.9x)	8725 (1078/1870)	16%
4 Stages	1.17ns	855 (3.0x)	11276 (3388/2112)	50%
8 Stages	0.80ns	1250 (4.4x)	17127 (8938/2612)	127%

- Pipeline efficiency
  - 2-stage: nearly double throughput; marginal area cost
  - 4-stage: 75% efficiency; area still reasonable
  - 8-stage: 55% efficiency; area more than doubles
- Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC

### Pipelining Idealisms

- Uniform subcomputations
  - Can pipeline into stages with equal delay
  - **Balance pipeline stages**
- Identical computations
  - Can fill pipeline with identical work
  - **Unify instruction types**
- Independent computations
  - No relationships between work units
  - **Minimize pipeline stalls**
- Are these practical?
  - No, but can get close enough to get significant speedup

### Instruction Pipelining

- The “computation” to be pipelined.
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - Operand(s) Fetch (OF)
  - Instruction Execution (EX)
  - Operand Store (OS)
  - Update Program Counter (PC)

### Generic Instruction Pipeline

- Based on “obvious” subcomputations

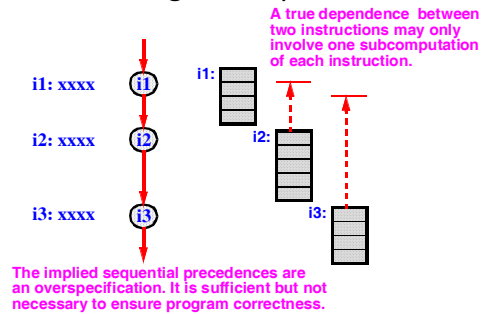
## Pipelining Idealisms

- ☑ Uniform subcomputations
  - Can pipeline into stages with equal delay
  - Balance pipeline stages
- ☑ Identical computations
  - Can fill pipeline with identical work
  - Unify instruction types (example in 752 notes)
- Independent computations
  - No relationships between work units
  - Minimize pipeline stalls

© 2005 Mikko Lipasti

13

## Program Dependences



© 2005 Mikko Lipasti

14

## Program Data Dependences

- True dependence (RAW)
  - j cannot execute until i produces its result
$$D(i) \cap R(j) \neq \phi$$
- Anti-dependence (WAR)
  - j cannot write its result until i has read its sources
$$R(i) \cap D(j) \neq \phi$$
- Output dependence (WAW)
  - j cannot write its result until i has written its result
$$D(i) \cap D(j) \neq \phi$$

© 2005 Mikko Lipasti

15

## Control Dependences

- Conditional branches
  - Branch must execute to determine which instruction to fetch next
  - Instructions following a conditional branch are control dependent on the branch instruction

© 2005 Mikko Lipasti

16

## Resolution of Pipeline Hazards

- Pipeline hazards
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard resolution
  - Static: compiler/programmer guarantees correctness
  - Dynamic: hardware performs checks at runtime
- Pipeline interlock
  - Hardware mechanism for dynamic hazard resolution
  - Must detect and enforce dependences at runtime

© 2005 Mikko Lipasti

17

## IBM RISC Experience

[Agerwala and Cocke 1987]

- Internal IBM study: Limits of a scalar pipeline?
- Memory Bandwidth
  - Fetch 1 instr/cycle from I-cache
  - 40% of instructions are load/store (D-cache)
- Code characteristics (dynamic)
  - Loads – 25%
  - Stores 15%
  - ALU/RR – 40%
  - Branches – 20%
    - 1/3 unconditional (always taken)
    - 1/3 conditional taken, 1/3 conditional not taken

© 2005 Mikko Lipasti

18

## IBM Experience

- Cache Performance
  - Assume 100% hit ratio (upper bound)
  - Cache latency:  $I = D = 1$  cycle default
- Load and branch scheduling
  - Loads
    - 25% cannot be scheduled (delay slot empty)
    - 65% can be moved back 1 or 2 instructions
    - 10% can be moved back 1 instruction
  - Branches
    - Unconditional – 100% schedulable (fill one delay slot)
    - Conditional – 50% schedulable (fill one delay slot)

© 2005 Mikko Lipasti

19

## CPI Optimizations

- Goal and impediments
  - $CPI = 1$ , prevented by pipeline stalls
- No cache bypass of RF, no load/branch scheduling
  - Load penalty: 2 cycles:  $0.25 \times 2 = 0.5$  CPI
  - Branch penalty: 2 cycles:  $0.2 \times 2/3 \times 2 = 0.27$  CPI
  - Total CPI:  $1 + 0.5 + 0.27 = 1.77$  CPI
- Bypass, no load/branch scheduling
  - Load penalty: 1 cycle:  $0.25 \times 1 = 0.25$  CPI
  - Total CPI:  $1 + 0.25 + 0.27 = 1.52$  CPI

© 2005 Mikko Lipasti

20

## More CPI Optimizations

- Bypass, scheduling of loads/branches
  - Load penalty:
    - $65\% + 10\% = 75\%$  moved back, no penalty
    - 25% => 1 cycle penalty
    - $0.25 \times 0.25 \times 1 = 0.0625$  CPI
  - Branch Penalty
    - 1/3 unconditional 100% schedulable => 1 cycle
    - 1/3 cond. not-taken, => no penalty (predict not-taken)
    - 1/3 cond. Taken, 50% schedulable => 1 cycle
    - 1/3 cond. Taken, 50% unschedulable => 2 cycles
    - $0.25 \times [1/3 \times 1 + 1/3 \times 0.5 \times 1 + 1/3 \times 0.5 \times 2] = 0.167$
- Total CPI:  $1 + 0.063 + 0.167 = 1.23$  CPI

© 2005 Mikko Lipasti

21

## Simplify Branches

- Assume 90% can be PC-relative
  - No register indirect, no register access
  - Separate adder (like MIPS R3000)
  - Branch penalty reduced
- Total CPI:  $1 + 0.063 + 0.085 = 1.15$  CPI = 0.87 IPC

15% Overhead from program dependences

PC-relative	Schedulable	Penalty
Yes (90%)	Yes (50%)	0 cycle
Yes (90%)	No (50%)	1 cycle
No (10%)	Yes (50%)	1 cycle
No (10%)	No (50%)	2 cycles

© 2005 Mikko Lipasti

22

## Limits of Pipelining

- IBM RISC Experience
  - Control and data dependences add 15%
  - Best case CPI of 1.15, IPC of 0.87
  - Deeper pipelines (higher frequency) magnify dependence penalties
- This analysis assumes 100% cache hit rates
  - Hit rates approach 100% for some programs
  - Many important programs have much worse hit rates

## Processor Performance

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size)                      (CPI)                      (cycle time)

- In the 1980's (decade of pipelining):
  - CPI: 5.0 => 1.15
- In the 1990's (decade of superscalar):
  - CPI: 1.15 => 0.5 (best case)
- In the 2000's (decade of multicore):
  - Core CPI unchanged; chip CPI scales with #cores

## Limits on Instruction Level Parallelism (ILP)

Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86 (Flynn's bottleneck)
Tjaden and Flynn [1973]	1.96
Uhi [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7 (Jouppi disagreed)
Kuck et al. [1972]	8
Riseman and Foster [1972]	51 (no control dependences)
Nicolau and Fisher [1984]	90 (Fisher's optimism)

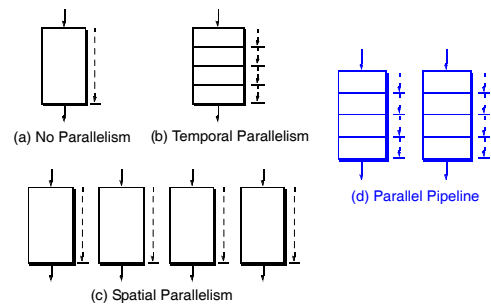
## Superscalar Proposal

- Go beyond single instruction pipeline, achieve  $IPC > 1$
- Dispatch multiple instructions per cycle
- Provide more generally applicable form of concurrency (not just vectors)
- Geared for sequential code that is hard to parallelize otherwise
- Exploit **fine-grained or instruction-level parallelism (ILP)**

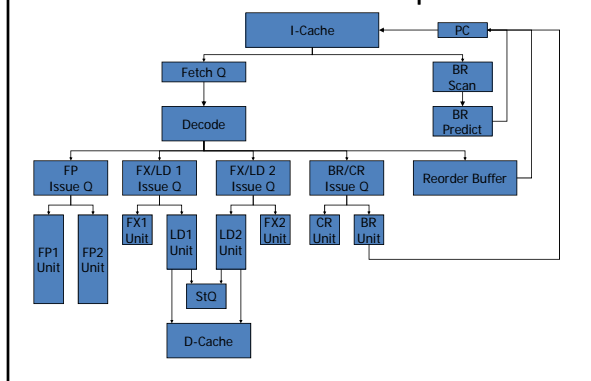
## Limitations of Scalar Pipelines

- Scalar upper bound on throughput
  - $IPC \leq 1$  or  $CPI \geq 1$
- Inefficient unified pipeline
  - Long latency for each instruction
- Rigid pipeline stall policy
  - One stalled instruction stalls all newer instructions

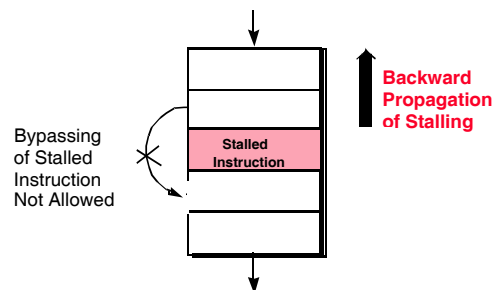
## Parallel Pipelines



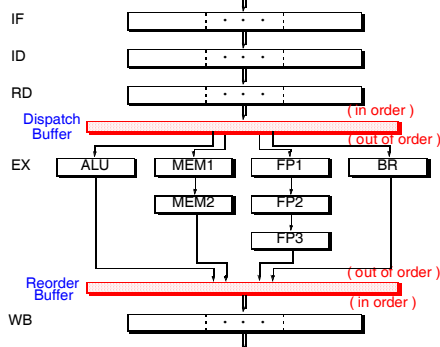
## Power4 Diversified Pipelines



## Rigid Pipeline Stall Policy



## Dynamic Pipelines



## Limitations of Scalar Pipelines

- Scalar upper bound on throughput
  - $IPC \leq 1$  or  $CPI \geq 1$
  - Solution: wide (superscalar) pipeline
- Inefficient unified pipeline
  - Long latency for each instruction
  - Solution: diversified, specialized pipelines
- Rigid pipeline stall policy
  - One stalled instruction stalls all newer instructions
  - Solution: Out-of-order execution, distributed execution pipelines

## Superscalar Overview

- Instruction flow
  - Branches, jumps, calls: predict target, direction
  - Fetch alignment
  - Instruction cache misses
- Register data flow
  - Register renaming: RAW/WAR/WAW
- Memory data flow
  - In-order stores: WAR/WAW
  - Store queue: RAW
  - Data cache misses

## Goal and Impediments

- Goal of Instruction Flow
  - Supply processor with maximum number of useful instructions every clock cycle
- Impediments
  - Branches and jumps
  - Finite I-Cache
    - Capacity
    - Bandwidth restrictions

## Limits on Instruction Level Parallelism (ILP)

Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86 (Flynn's bottleneck)
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7 (Jouppi disagreed)
Kuck et al. [1972]	8
Riseman and Foster [1972]	51 (no control dependences)
Nicolau and Fisher [1984]	90 (Fisher's optimism)

## Speculative Execution

- Riseman & Foster showed potential
  - But no idea how to reap benefit
- 1979: Jim Smith patents branch prediction at Control Data
  - Predict current branch based on past history
- Today: virtually all processors use branch prediction

## Improving I-Cache Performance

- Larger cache size
  - Code compression
  - Instruction registers
- Increased associativity
  - Conflict misses less of a problem than in data caches
- Larger line size
  - Spatial locality inherent in sequential program I-stream
- Code layout
  - Maximize instruction stream's spatial locality
- Cache prefetching
  - Next-line, streaming buffer
  - Branch target (even if not taken)
- Other types of I-cache organization
  - Trace cache [Ch. 9]

© 2005 Mikko Lipasti

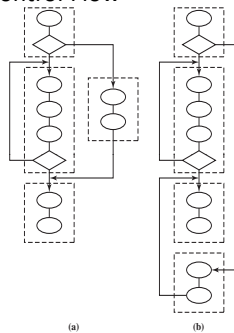
37

## Program Control Flow

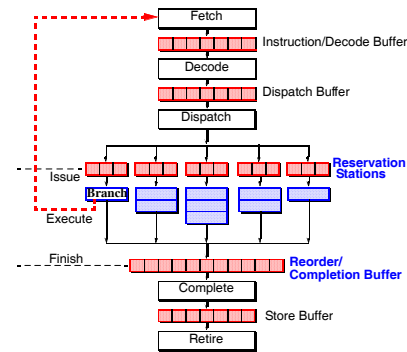
- Implicit Sequential Control Flow
  - Static Program Representation
    - Control Flow Graph (CFG)
    - Nodes = basic blocks
    - Edges = Control flow transfers
  - Physical Program Layout
    - Mapping of CFG to linear program memory
    - Implied sequential control flow
  - Dynamic Program Execution
    - Traversal of the CFG nodes and edges (e.g. loops)
    - Traversal dictated by branch conditions
  - Dynamic Control Flow
    - Deviates from sequential control flow
    - Disrupts sequential fetching
    - Can stall IF stage and reduce I-fetch bandwidth

## Program Control Flow

- Dynamic traversal of static CFG
- Mapping CFG to linear memory



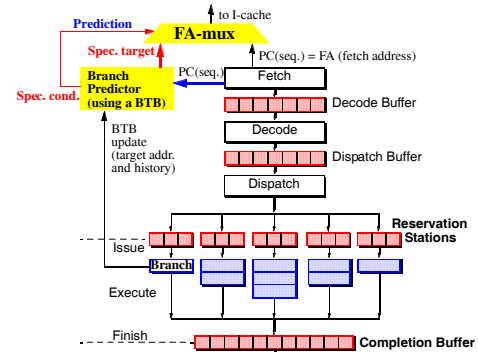
## Disruption of Sequential Control Flow



## Branch Prediction

- Target address generation → Target Speculation
  - Access register:
    - PC, General purpose register, Link register
  - Perform calculation:
    - +/- offset, autoincrement, autodecrement
- Condition resolution → Condition speculation
  - Access register:
    - Condition code register, General purpose register
  - Perform calculation:
    - Comparison of data register(s)

## Branch Instruction Speculation



### Branch/Jump Target Prediction

0x0348	0101 (NTNT)	0x0612

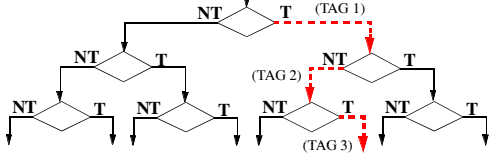
Branch Inst. address      Information for predict.      Branch target address (most recent)

- **Branch Target Buffer:** small cache in fetch stage
  - Previously executed branches, address, taken history, target(s)
- Fetch stage compares current FA against BTB
  - If match, use prediction
  - If predict taken, use BTB target
- When branch executes, BTB is updated
- Optimization:
  - Size of BTB: increases hit rate
  - Prediction algorithm: increase accuracy of prediction

### Branch Prediction: Condition Speculation

1. **Biased Not Taken**
  - Hardware prediction
  - Does not affect ISA
  - Not effective for loops
2. **Software Prediction**
  - Extra bit in each branch instruction
    - Set to 0 for not taken
    - Set to 1 for taken
  - Bit set by compiler or user; can use profiling
  - Static prediction, same behavior every time
3. **Prediction based on branch offset**
  - Positive offset: predict not taken
  - Negative offset: predict taken
4. **Prediction based on dynamic history**

### Branch Speculation

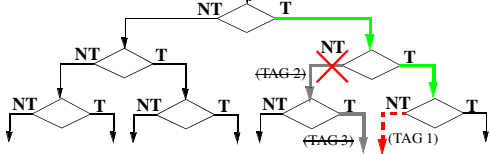


- **Leading Speculation**
  - Typically done during the Fetch stage
  - Based on potential branch instruction(s) in the current fetch group
- **Trailing Confirmation**
  - Typically done during the Branch Execute stage
  - Based on the next Branch instruction to finish execution

### Branch Speculation

- **Leading Speculation**
  1. Tag speculative instructions
  2. Advance branch and following instructions
  3. Buffer addresses of speculated branch instructions
- **Trailing Confirmation**
  1. When branch resolves, remove/deallocate speculation tag
  2. Permit completion of branch and following instructions

### Branch Speculation



- **Start new correct path**
  - Must remember the alternate (non-predicted) path
- **Eliminate incorrect path**
  - Must ensure that the mis-speculated instructions produce no side effects

### Mis-speculation Recovery

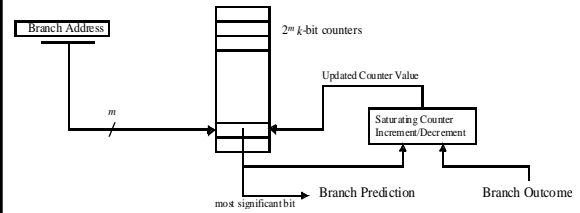
- **Start new correct path**
  1. Update PC with computed branch target (if predicted NT)
  2. Update PC with sequential instruction address (if predicted T)
  3. Can begin speculation again at next branch
- **Eliminate incorrect path**
  1. Use tag(s) to deallocate ROB entries occupied by speculative instructions
  2. Invalidate all instructions in the decode and dispatch buffers, as well as those in reservation stations



## Dynamic Branch Prediction

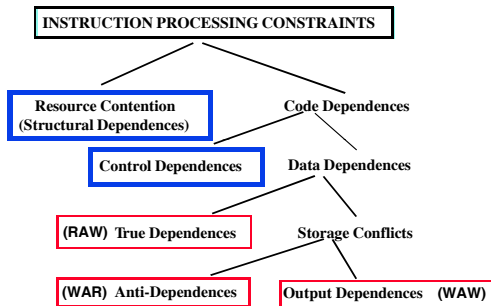
- Main advantages:
  - Learn branch behavior autonomously
    - No compiler analysis, heuristics, or profiling
  - Adapt to changing branch behavior
    - Program phase changes branch behavior
- First proposed in 1980
  - US Patent #4,370,711, Branch predictor using random access memory, James. E. Smith
- Continually refined since then

## Smith Predictor Hardware



- Jim E. Smith. A Study of Branch Prediction Strategies. International Symposium on Computer Architecture, pages 135-148, May 1981
- Widely employed: Intel Pentium, PowerPC 604, PowerPC 620, etc.

## The Big Picture



© Shen, Lipasti

51

## Register Data Dependences

- Program data dependences cause hazards
  - True dependences (RAW)
  - Antidependences (WAR)
  - Output dependences (WAW)
- When are registers read and written?
  - Out of program order!
  - Hence, any/all of these can occur
- Solution to all three: **register renaming**

© Shen, Lipasti

52

## Register Renaming: WAR/WAW

- Widely employed (P-4, P-M, Athlon, ...)
- Resolving WAR/WAW:
  - Each register write gets unique “**rename register**”
  - Writes are **committed in program order** at Writeback
  - WAR and WAW are not an issue
    - All updates to “architected state” delayed till writeback
    - **Writeback stage always later than read stage**
  - **Reorder Buffer (ROB)** enforces in-order writeback

Add R3 <= ...	P32 <= ...
Sub R4 <= ...	P33 <= ...
And R3 <= ...	P35 <= ...

© Shen, Lipasti

53

## Register Renaming: RAW

- In order, at dispatch:
  - Source registers checked to see if “in flight”
    - Register map table keeps track of this
    - If not in flight, read from real register file
    - If in flight, look up “rename register” tag (IOU)
  - Then, allocate new register for register write

Add R3 <= R2 + R1	P32 <= P2 + P1
Sub R4 <= R3 + R1	P33 <= P32 + P1
And R3 <= R4 & R2	P35 <= P33 + P2

© Shen, Lipasti

54

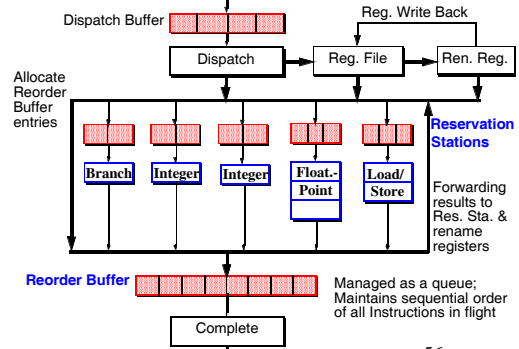
## Register Renaming: RAW

- Advance instruction to reservation station
  - Wait for rename register tag to trigger issue
- Reservation station enables out-of-order issue
  - Newer instructions can bypass stalled instructions

© Shen, Lipasti

55

## “Dataflow Engine” for Dynamic Execution



© Shen, Lipasti

56

## Instruction Processing Steps

### •DISPATCH:

- Read operands from Register File (RF) and/or Rename Buffers (RRB)
- Rename destination register and allocate RRF entry
- Allocate Reorder Buffer (ROB) entry
- Advance instruction to appropriate Reservation Station (RS)

### •EXECUTE:

- RS entry monitors bus for register Tag(s) to latch in pending operand(s)
- When all operands ready, issue instruction into Functional Unit (FU) and deallocate RS entry (no further stalling in execution pipe)
- When execution finishes, broadcast result to waiting RS entries, RRB entry, and ROB entry

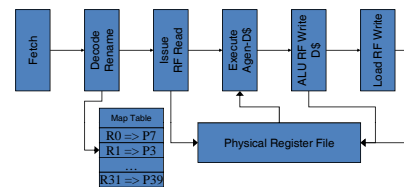
### •COMPLETE:

- Update architected register from RRB entry, deallocate RRB entry, and if it is a store instruction, advance it to Store Buffer
- Deallocate ROB entry and instruction is considered architecturally completed

© Shen, Lipasti

57

## Physical Register File



- Used in the MIPS R10000 pipeline
- All registers in one place
  - Always accessed right before EX stage
  - No copying to real register file

© Shen, Lipasti

58

## Managing Physical Registers

Map Table
R0 => P7
R1 => P3
...
R31 => P39

Add R3 <= R2 + R1    P32 <= P2 + P1  
 Sub R4 <= R3 + R1    P33 <= P32 + P1  
 ...  
 And R3 <= R4 & R2    P35 <= P33 + P2

Release P32 (previous R3) when this instruction completes execution

- What to do when all physical registers are in use?
  - Must release them somehow to avoid stalling
  - Maintain *free list* of “unused” physical registers
- Release when no more uses are possible
  - Sufficient: next write commits

© Shen, Lipasti

59

## Memory Data Dependences

- Besides branches, long memory latencies are one of the biggest performance challenges today.
- To preserve sequential (in-order) state in the data caches and external memory (so that recovery from exceptions is possible) **stores are performed in order**. This takes care of antidependences and output dependences to memory locations.
- However, **loads can be issued out of order** with respect to stores if the out-of-order loads check for data dependences with respect to previous, pending stores.



### Memory Data Dependences

- **"Memory Aliasing"** = Two memory references involving the same memory location (collision of two memory addresses).
- **"Memory Disambiguation"** = Determining whether two memory references will alias or not (whether there is a dependence or not).
- **Memory Dependency Detection:**
  - Must compute effective addresses of both memory references
  - Effective addresses can depend on run-time data and other instructions
  - Comparison of addresses require much wider comparators

Example code:

```

(1) STORE V
(2) ADD
(3) LOAD X
(4) LOAD X
(5) LOAD V
(6) ADD
(7) STORE W
  
```

### Memory Data Dependences

- WAR/WAW: stores commit in order
  - Hazards not possible.
- RAW: loads must check pending stores
  - Store queue keeps track of pending store addresses
  - Loads check against these addresses
  - Similar to register bypass logic
  - Comparators are 32 or 64 bits wide (address size)
- Major source of complexity in modern designs
  - Store queue lookup is position-based
  - What if store address is not yet known? Stall all trailing ops

© Shen, Lipasti 62

### Optimizing Load/Store Disambiguation

- Non-speculative load/store disambiguation
  1. Loads wait for addresses of all prior stores
  2. Full address comparison
  3. Bypass if no match, forward if match
- (1) can limit performance:

```

load r5, MEM[r3] ← cache miss
store r7, MEM[r5] ← RAW for agen, stalled
...
load r8, MEM[r9] ← independent load stalled
  
```

### Speculative Disambiguation

- What if aliases are rare?
  1. Loads don't wait for addresses of all prior stores
  2. Full address comparison of stores that are ready
  3. Bypass if no match, forward if match
  4. Check all store addresses when they commit
    - No matching loads - speculation was correct
    - Matching unbypassed load - incorrect speculation
  5. Replay starting from incorrect load

### Speculative Disambiguation: Load Bypass

```

i1: st R3, MEM[R8]: ??
i2: ld R9, MEM[R4]: ??
  
```

- i1 and i2 issue in program order
- i2 checks store queue (no match)

### Speculative Disambiguation: Load Forward

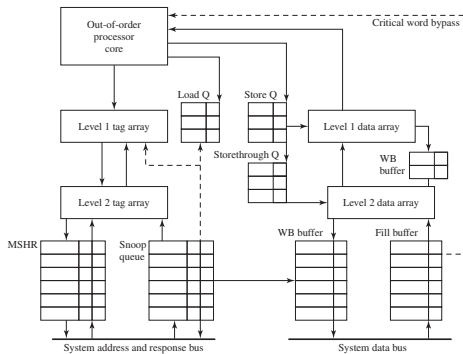
```

i1: st R3, MEM[R8]: ??
i2: ld R9, MEM[R4]: ??
  
```

- i1 and i2 issue in program order
- i2 checks store queue (match=>forward)



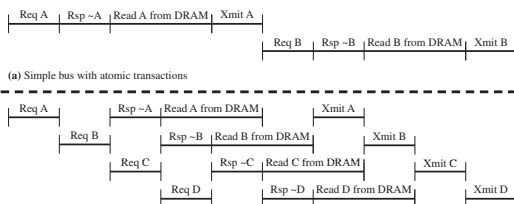
## Coherent Memory Interface



## Coherent Memory Interface

- **Load Queue**
  - Tracks inflight loads for aliasing, coherence
- **Store Queue**
  - Defers stores until commit, tracks aliasing
- **Storethrough Queue or Write Buffer or Store Buffer**
  - Defers stores, coalesces writes, must handle RAW
- **MSHR**
  - Tracks outstanding misses, enables *lockup-free caches* [Kroft ISCA 91]
- **Snoop Queue**
  - Buffers, tracks incoming requests from coherent I/O, other processors
- **Fill Buffer**
  - Works with MSHR to hold incoming partial lines
- **Writeback Buffer**
  - Defers writeback of evicted line (demand miss handled first)

## Split Transaction Bus



- "Packet switched" vs. "circuit switched"
- Release bus after request issued
- Allow multiple concurrent requests to overlap memory latency
- Complicates control, arbitration, and coherence protocol
  - *Transient* states for pending blocks (e.g. "req. issued but not completed")

## Memory Consistency

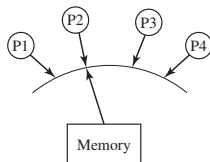
```

Reorder load before store
Proc0
if (load B==0) {
    ...critical section
}

Proc1
st B=1
if (load A==0) {
    ...critical section
}
    
```

- How are memory references from different processors interleaved?
- If this is not well-specified, synchronization becomes difficult or even impossible
  - ISA must specify consistency model
- Common example using Dekker's algorithm for synchronization
  - If load reordered ahead of store (as we assume for a baseline OOO CPU)
  - Both Proc0 and Proc1 enter critical section, since both observe that other's lock variable (A/B) is not set
- If consistency model allows loads to execute ahead of stores, Dekker's algorithm no longer works
  - Common ISAs allow this: IA-32, PowerPC, SPARC, Alpha

## Sequential Consistency [Lamport 1979]

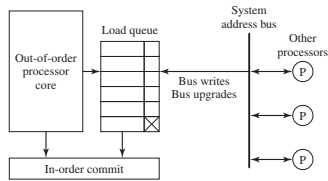


- Processors treated as if they are interleaved processes on a single time-shared CPU
- All references must fit into a total global order or interleaving that does not violate any CPUs program order
  - Otherwise sequential consistency not maintained
- Now Dekker's algorithm will work
- Appears to preclude any OOO memory references
  - Hence precludes any real benefit from OOO CPUs

## High-Performance Sequential Consistency

- Coherent caches isolate CPUs if no sharing is occurring
  - Absence of coherence activity means CPU is free to reorder references
- Still have to order references with respect to misses and other coherence activity (snoops)
- Key: use speculation
  - Reorder references speculatively
  - Track which addresses were touched speculatively
  - Force replay (in order execution) of such references that collide with coherence activity (snoops)

### High-Performance Sequential Consistency



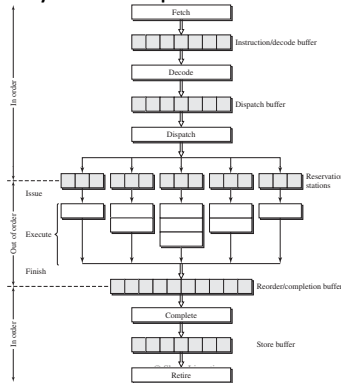
- Load queue records all speculative loads
- Bus writes/upgrades are checked against LQ
- Any matching load gets marked for replay
- At commit, loads are checked and replayed if necessary
  - Results in machine flush, since load-dependent ops must also replay
- Practically, conflicts are rare, so expensive flush is OK

### Issues in Completion/Retirement

- Out-of-order execution
  - ALU instructions
  - Load/store instructions
- In-order completion/retirement
  - Precise exceptions
  - Memory ordering in multiprocessors
- Solutions
  - Reorder buffer retires instructions in order
  - Store queue retires stores in order
  - Exceptions can be handled at any instruction boundary by reconstructing state out of ROB/SQ
  - Load queue monitors other processors' references

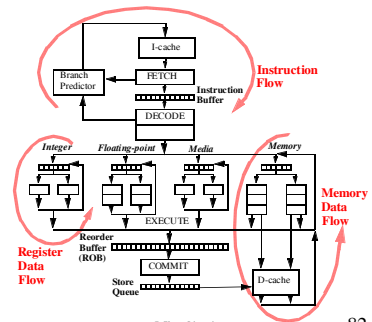
© Shen, Lipasti 80

### A Dynamic Superscalar Processor

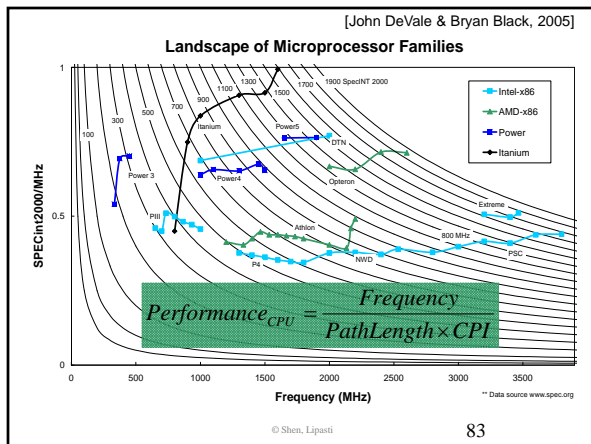


81

### Superscalar Summary



© Shen, Lipasti 82



© Shen, Lipasti 83

### Review of 752

- ✓ Iron law
- ✓ Beyond pipelining
- ✓ Superscalar challenges
  - ✓ Instruction flow
  - ✓ Register data flow
  - ✓ Memory Dataflow
- ✓ Modern memory interface
- What was not covered
  - Memory hierarchy (caches, DRAM)
  - Virtual memory
  - Power
  - Many implementation/design details
  - Etc.
  - Multithreading (next week)