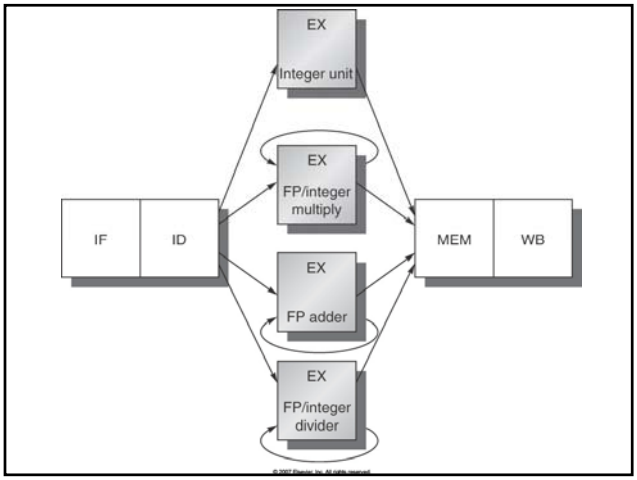
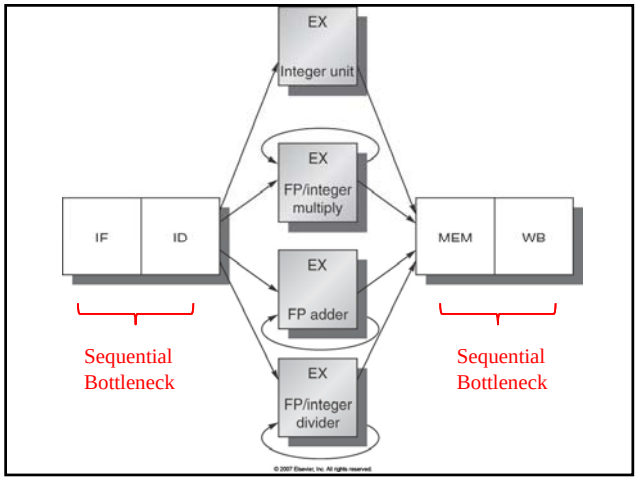
Superscalar Organization

HPCA

Spring 2011

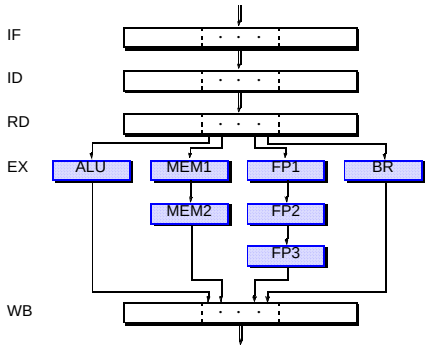




Limitations of Scalar Pipelines

- Scalar upper bound on throughput
  - IPC <= 1 or CPI >= 1
- Inefficient unified pipeline
  - Long latency for each instruction
- Rigid pipeline stall policy
  - One stalled instruction stalls all newer instructions
  - Tomasulo's algorithm alleviated this problem

Superscalar Pipeline



Superscalar Pipeline

- Fetch & Decode multiple instructions per cycle
- Multiple (diversified) functional units for instruction execution
- Register renaming and out-of-order issue (like Tomasulo's algorithm)

BUT…

 6

Challenge for Superscalar Pipes

- How to keep the pipeline operating at or near full capacity?
  - Wide instruction fetch gobbles up instructions at a high rate
  - Branches are encountered frequently
  - Cost of stalls is much higher than for scalar pipelines
- Branches pose the biggest challenge to exploiting Instruction Level Parallelism (ILP)

 7

Superscalar Pipelines—Exploiting ILP

- To maintain a steady stream of instructions to feed functional units it is necessary to maintain instruction fetch and execution beyond branch points
- This leads to "speculative execution" of instructions
  - Accurate branch prediction is essential
  - Must insure that wrong guesses don't lead to incorrect behavior

 8

Speculation for greater ILP

- Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct
  - Speculation ⇒ fetch, issue, and execute instructions as if branch predictions were always correct
  - Dynamic scheduling ⇒ only fetches and issues instructions
- Essentially a data flow execution model: Operations execute as soon as their operands are available

9

Speculation for greater ILP

- 3 components of HW-based speculation:

1. Dynamic branch prediction to choose which instructions to execute
2. Speculation to allow execution of instructions before control dependences are resolved
   + ability to undo effects of incorrectly speculated sequence
3. Dynamic scheduling to deal with scheduling of different combinations of basic blocks

10

Adding Speculation to Tomasulo's Algorithm

- Must separate execution from instruction completion or "commit"
- This additional step called instruction commit
- When an instruction is no longer speculative, allow it to update the register file or memory
- Requires additional set of buffers to hold results of instructions that have finished execution but have not committed
- This reorder buffer (ROB) is also used to pass results among instructions that may be speculated

11

Reorder Buffer (ROB)

- In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
- The ROB supplies operands in interval between completion of instruction execution and instruction commit
  - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo's algorithm
  - ROB extends architected registers like Reservation Stations

12

Reorder Buffer Entry

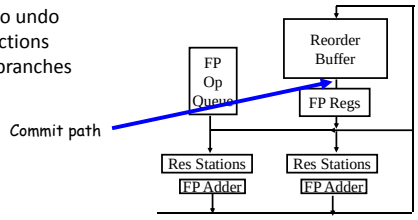
- Each entry in the ROB contains four fields:

1. Instruction type
   - a branch (has no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, which has register destinations)
2. Destination
   - Register number (for loads and ALU operations) or memory address (for stores) where the instruction result should be written
3. Value
   - Value of instruction result until the instruction commits
4. Ready
   - Indicates that instruction has completed execution, and the value is ready

13

Reorder Buffer operation

- Holds instructions in FIFO order, exactly as dispatched
- When instructions complete, results placed into ROB
  - Supplies operands to other instruction between execution complete & commit ⇒ more registers like RS
  - Tag results with ROB buffer number instead of reservation station
- Instructions commit ⇒values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions



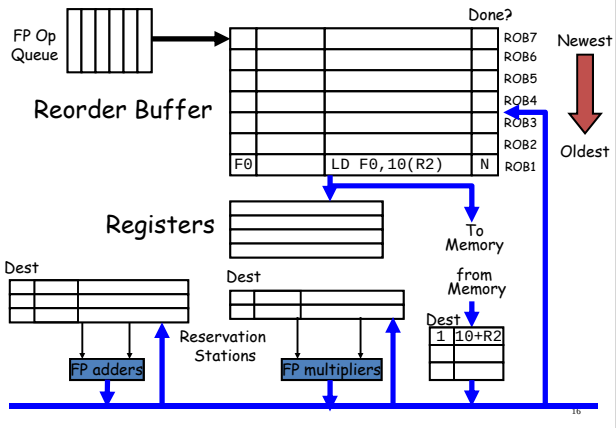
14

Four Steps of Speculative Tomasulo' Algorithm

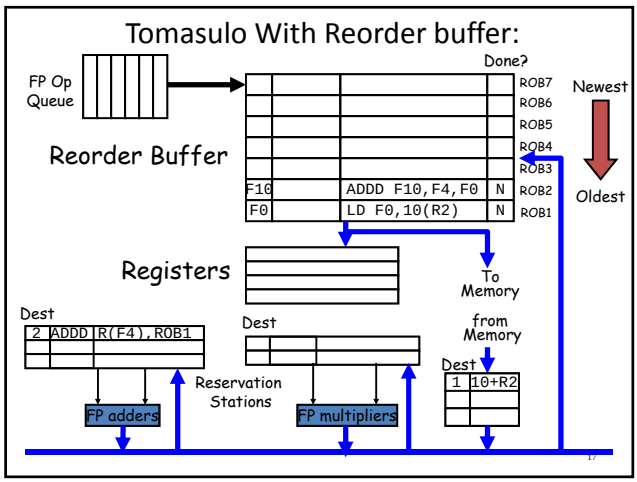
1. Issue—get instruction from FP Op Queue
   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")
2. Execution—operate on operands (EX)
   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called "issue")
3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.
4. Commit—update register with reorder result
   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")

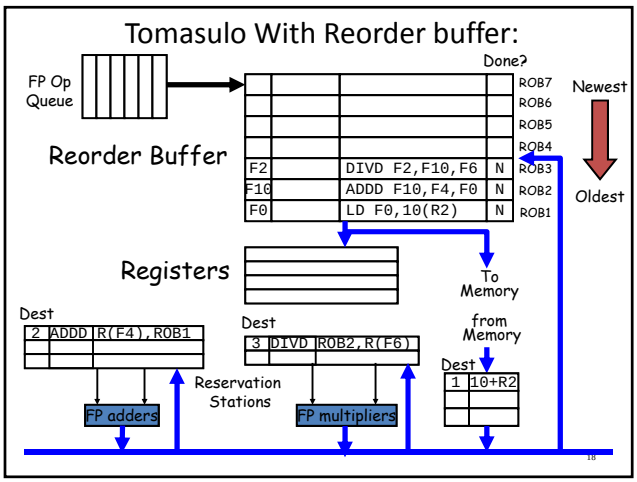
15

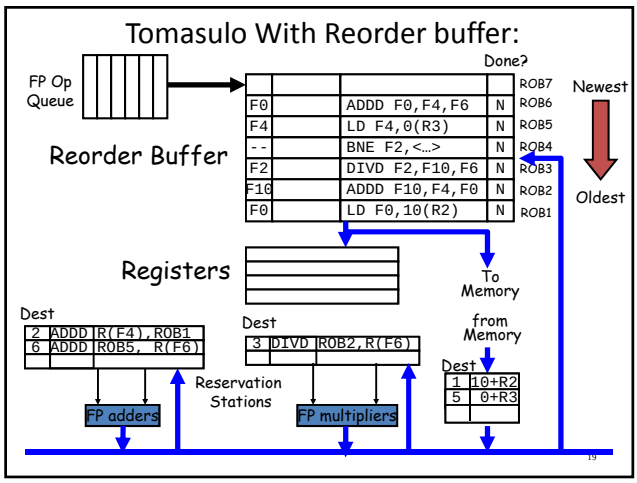
Tomasulo With Reorder buffer:

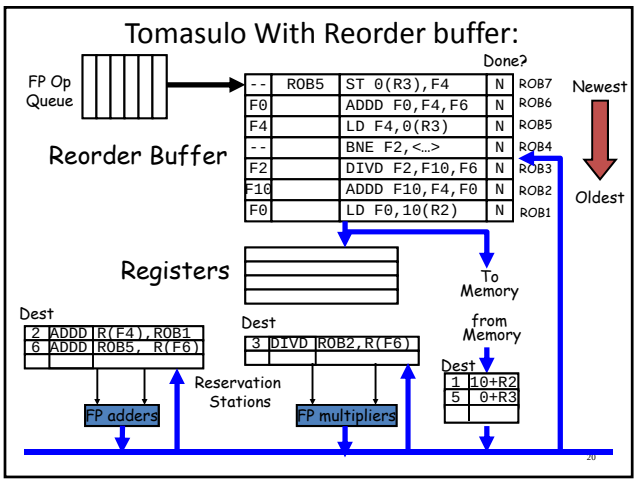


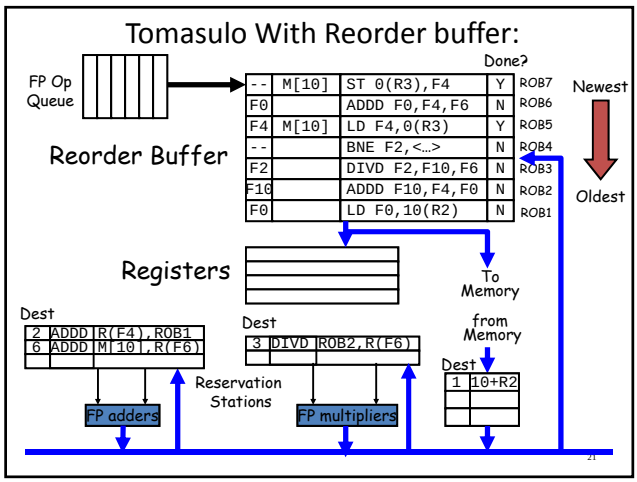
16

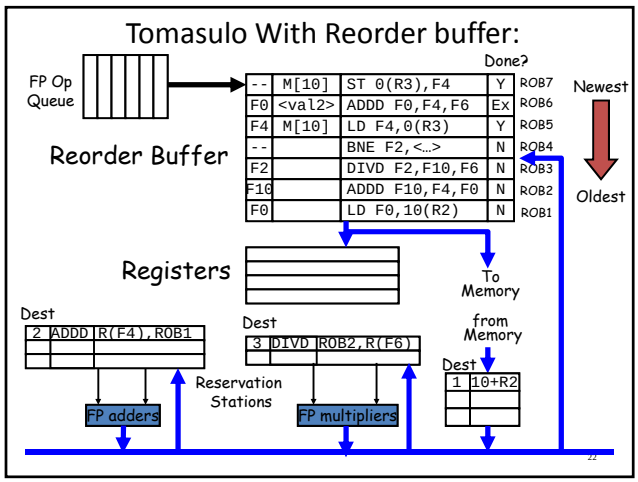

Tomasulo With Reorder buffer:
FP Op Queue
Done?
Newest
Oldest
Reorder Buffer
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
F10 ADDD F10,F4,F0 N
F0 LD F0,10(R2) N
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
Dest
Dest
1 10+R2
Reservation Stations
FP adders
FP multipliers

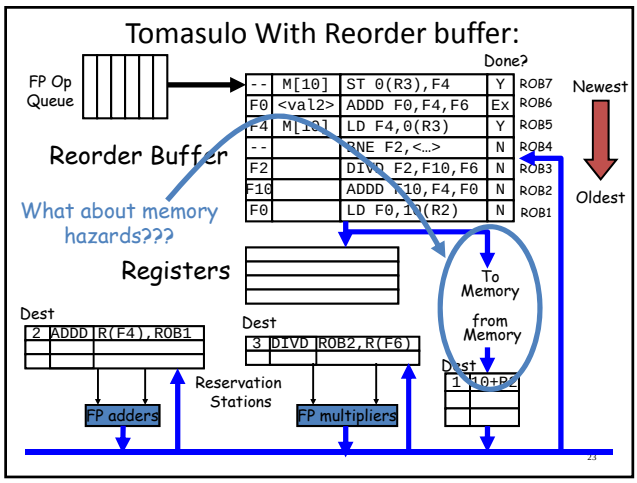

Tomasulo With Reorder buffer:
FP Op Queue
Done?
Newest
Oldest
Reorder Buffer
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
F2 DIVD F2,F10,F6 N
F10 ADDD F10,F4,F0 N
F0 LD F0,10(R2) N
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
Dest
3 DIVD ROB2,R(F6)
Dest
1 10+R2
Reservation Stations
FP adders
FP multipliers


Tomasulo With Reorder buffer:
FP Op Queue
Done?
Newest
Oldest
Reorder Buffer
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
F0 ADDD F0,F4,F6 N
F4 LD F4,0(R3) N
-- BNE F2,<…> N
F2 DIVD F2,F10,F6 N
F10 ADDD F10,F4,F0 N
F0 LD F0,10(R2) N
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
6 ADDD ROB5, R(F6)
Dest
3 DIVD ROB2,R(F6)
Dest
1 10+R2
5 0+R3
Reservation Stations
FP adders
FP multipliers


Tomasulo With Reorder buffer:
FP Op Queue
Done?
Newest
Oldest
Reorder Buffer
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
-- ROB5 ST 0(R3),F4 N
F0 ADDD F0,F4,F6 N
F4 LD F4,0(R3) N
-- BNE F2,<…> N
F2 DIVD F2,F10,F6 N
F10 ADDD F10,F4,F0 N
F0 LD F0,10(R2) N
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
6 ADDD ROB5, R(F6)
Dest
3 DIVD ROB2,R(F6)
Dest
1 10+R2
5 0+R3
Reservation Stations
FP adders
FP multipliers



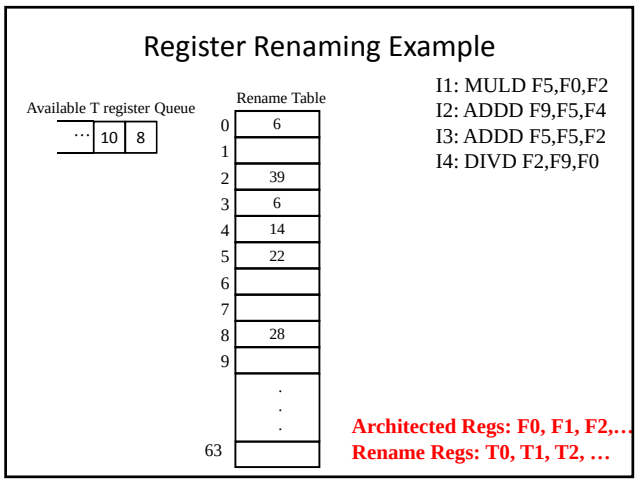


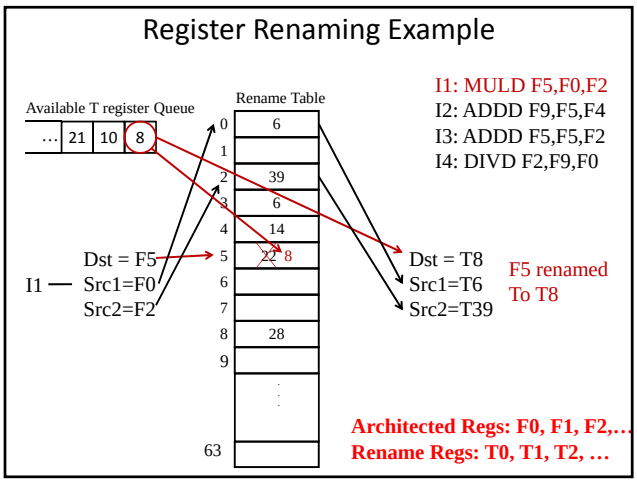


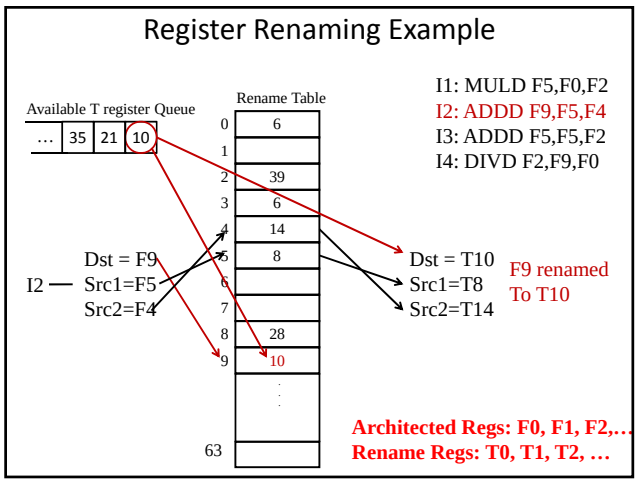
Speculation: Register Renaming vs. ROB

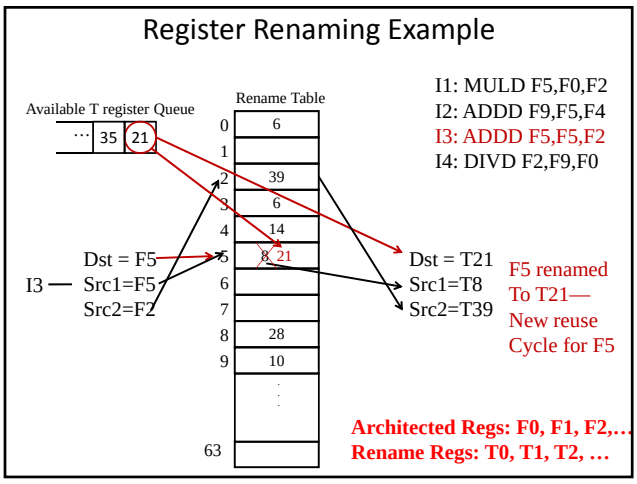
- Alternative to ROB is a larger physical set of registers combined with register renaming
  - Extended registers replace function of both ROB and reservation stations
- Instruction issue maps names of architectural registers to physical register numbers in extended register set
  - On issue, allocates a new unused register for the destination (which avoids WAW and WAR hazards)
  - Speculation recovery easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits
- Most Out-of-Order processors today use extended registers with renaming

24


Register Renaming Example
Available T register Queue
… 10 8
Rename Table
0 6
1
2 39
3 6
4 14
5 22
6
7
8 28
9
63
I1: MULD F5,F0,F2
I2: ADDD F9,F5,F4
I3: ADDD F5,F5,F2
I4: DIVD F2,F9,F0
Architected Regs: F0, F1, F2,…
Rename Regs: T0, T1, T2, …


Register Renaming Example
Available T register Queue
… 21 10 8
Rename Table
0 6
1
2 39
3 6
4 14
5 22 8
6
7
8 28
9
63
I1: MULD F5,F0,F2
I2: ADDD F9,F5,F4
I3: ADDD F5,F5,F2
I4: DIVD F2,F9,F0
I1
Dst = F5
Src1=F0
Src2=F2
Dst = T8
Src1=T6
Src2=T39
F5 renamed
To T8
Architected Regs: F0, F1, F2,…
Rename Regs: T0, T1, T2, …


Register Renaming Example
Available T register Queue
… 35 21 10
Rename Table
0 6
1
2 39
3 6
4 14
5 8
6
7
8 28
9 10
63
I1: MULD F5,F0,F2
I2: ADDD F9,F5,F4
I3: ADDD F5,F5,F2
I4: DIVD F2,F9,F0
I2
Dst = F9
Src1=F5
Src2=F4
Dst = T10
Src1=T8
Src2=T14
F9 renamed
To T10
Architected Regs: F0, F1, F2,…
Rename Regs: T0, T1, T2, …

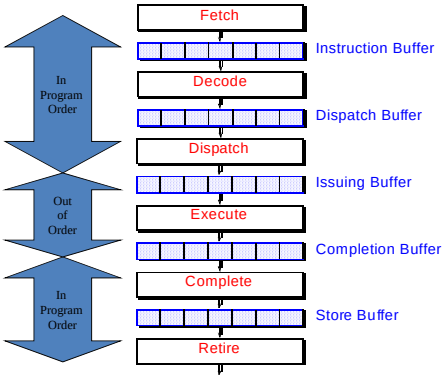

Register Renaming Example
Available T register Queue
… 35 21
Rename Table
0 6
1
2 39
3 6
4 14
5 8 21
6
7
8 28
9 10
63
I1: MULD F5,F0,F2
I2: ADDD F9,F5,F4
I3: ADDD F5,F5,F2
I4: DIVD F2,F9,F0
I3
Dst = F5
Src1=F5
Src2=F2
Dst = T21
Src1=T8
Src2=T39
F5 renamed
To T21—
New reuse
Cycle for F5
Architected Regs: F0, F1, F2,…
Rename Regs: T0, T1, T2, …

Register Renaming Notes

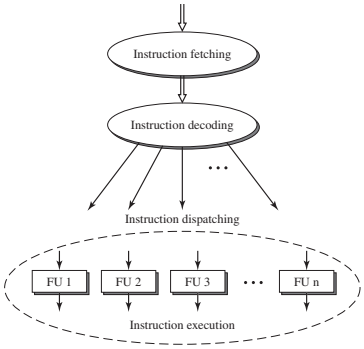
- Number of Rename Registers (T-Regs) can be larger than number of architected registers
  - This can alleviate the resource bottleneck for ISAs with small number of architected registers—e.g. Intel IA-32
- In some cases a result never needs to be written back to the architected register—like Tomasulo's algorithm
- A T-Reg is returned to Available T-reg Queue when the instruction that targets it commits

29

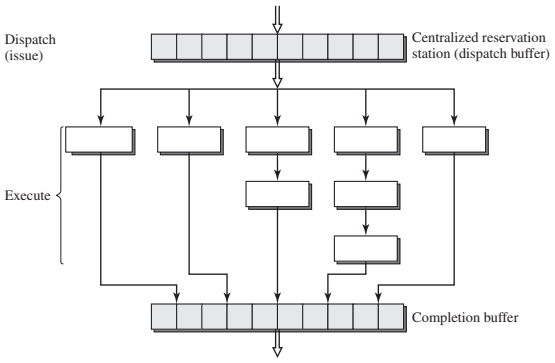
Superscalar Pipeline Stages



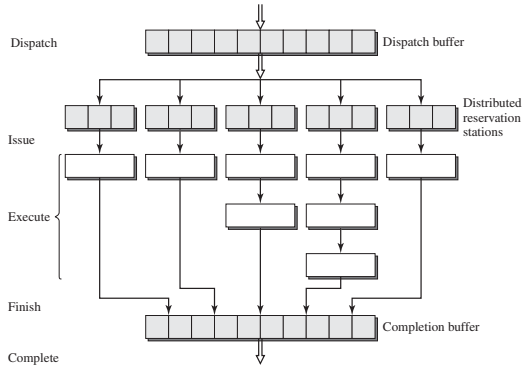
Necessity of Instruction Dispatch



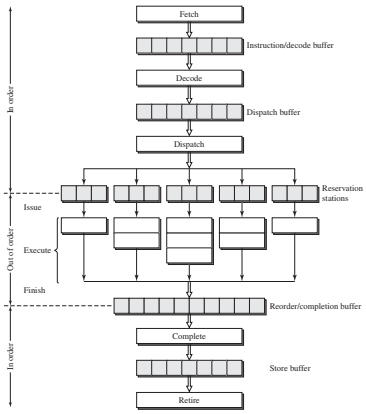
Centralized Reservation Station



Distributed Reservation Station



A Dynamic Superscalar Processor



Avoiding Memory Hazards

- WAW and WAR hazards through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending
- RAW hazards through memory are maintained by two restrictions:
  1. not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and
  2. maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.
- these restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data

35

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

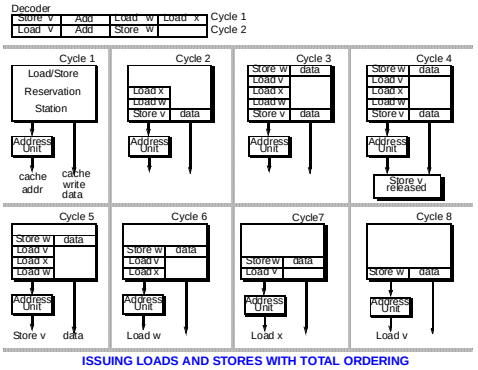
36

Conservative Approach: Maintain Total Order of Loads and Stores

- **Keep all loads and stores totally in order with respect to each other.**
- **However, loads and stores can execute out of order with respect to other types of instructions.**
- **Consequently, stores are held for all previous instructions, and loads are held for stores.**
  - **I.e. stores performed at commit point**
  - **Sufficient to prevent wrong branch path stores since all prior branches now resolved**

37

Illustration of Total Order



ISSUING LOADS AND STORES WITH TOTAL ORDERING

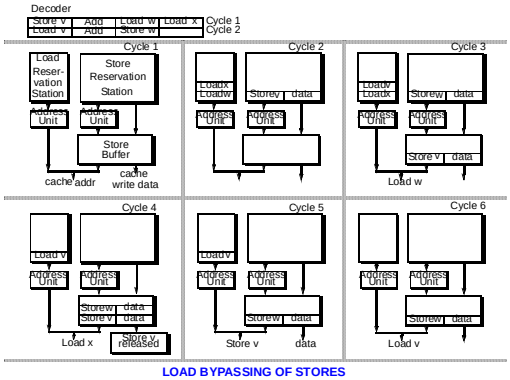
38

Load Bypassing

- **Loads can be allowed to bypass stores (if no aliasing).**
- **Two separate reservation stations and address generation units are employed for loads and stores.**
- **Store addresses still need to be computed before loads can be issued to allow checking for load dependences. If dependence cannot be checked, e.g. store address cannot be determined, then all subsequent loads are held until address is valid (conservative).**
- **Stores are kept in ROB until all previous instructions complete; and kept in the store buffer until gaining access to cache port.**

39

Illustration of Load Bypassing



LOAD BYPASSING OF STORES

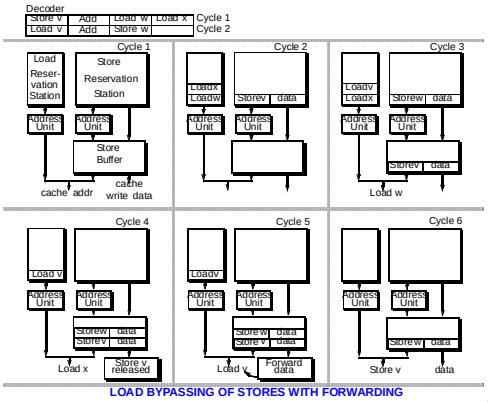
40

Load Forwarding

- **If a subsequent load has a dependence on a store still in the store buffer, it need not wait till the store is issued to the data cache.**
- **The load can be directly satisfied from the store buffer if the address is valid and the data is available in the store buffer.**
- **This avoids the latency of accessing the data cache.**

41

Illustration of Load Forwarding



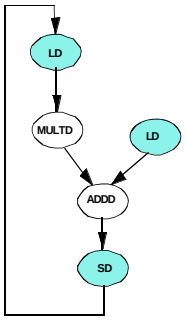
LOAD BYPASSING OF STORES WITH FORWARDING

42

The DAXPY Example

Y(i) = A * X(i) + Y(i)

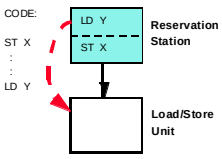
```
        LD      F0, a
        ADDI    R4, Rx, #512        ; last address
Loop:
        LD      F2, 0(Rx)           ; load X(i)
        MULTD   F2, F0, F2          ; A*X(i)
        LD      F4, 0(Ry)           ; load Y(i)
        ADDD    F4, F2, F4          ; A*X(i) + Y(i)
        SD      F4, 0(Ry)           ; store into Y(i)
        ADDI    Rx, Rx, #8          ; inc. index to X
        ADDI    Ry, Ry, #8          ; inc. index to Y
        SUB     R20, R4, Rx         ; compute bound
        BNZ     R20, loop           ; check if done
```



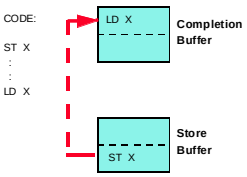
43

Performance Gains From Weak Ordering

Load Bypassing:



Load Forwarding:



Performance gain:

Load bypassing: 11%-19% increase over total ordering

Load forwarding: 1%-4% increase over load bypassing

44

Optimizing Load/Store Disambiguation

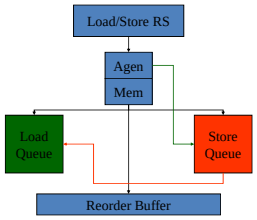
- Non-speculative load/store disambiguation
  1. Loads wait for addresses of all prior stores
  2. Full address comparison
  3. Bypass if no match, forward if match
- (1) can limit performance:

```
load r5,MEM[r3]     ← cache miss
store r7, MEM[r5]   ← RAW for address generation, stalled
…
load r8, MEM[r9]    ← independent load stalled
```

45

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



46

Use of Prediction

- If aliases are rare: static prediction
  - Predict no alias every time
    - Why even implement forwarding? PowerPC 620 doesn't
  - Pay misprediction penalty rarely
- If aliases are more frequent: dynamic prediction
  - Use PHT-like history table for loads
    - If alias predicted: delay load
    - If aliased pair predicted: forward from store to load
      - More difficult to predict pair [store sets, Alpha 21264]
  - Pay misprediction penalty rarely
- Memory cloaking [Moshovos, Sohi]
  - Predict load/store pair
  - Directly copy store data register to load target register
  - Reduce data transfer latency to absolute minimum

47

Load/Store Disambiguation Discussion

- RISC ISA:
  - Many registers, most variables allocated to registers
  - Aliases are rare
  - Most important to not delay loads (bypass)
  - Alias predictor may/may not be necessary
- CISC ISA:
  - Few registers, many operands from memory
  - Aliases much more common, forwarding necessary
  - Incorrect load speculation should be avoided
  - If load speculation allowed, predictor probably necessary
- Address translation:
  - Can't use virtual address (must use physical)
  - Wait till after TLB lookup is done
  - Or, use subset of untranslated bits (page offset)
    - Safe for proving inequality (bypassing OK)
    - Not sufficient for showing equality (forwarding not OK)

48

Exceptions and Interrupts

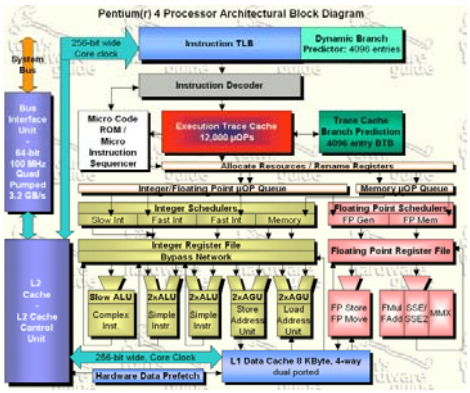
- IBM 360/91 invented "imprecise interrupts"
  - Computer stopped at this PC; its likely close to this address
  - Not so popular with programmers
- Technique for both precise interrupts/exceptions and speculation: in-order completion and in-order commit
  - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
  - This is exactly same as need to do with precise exceptions
- Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB
- If a speculated instruction raises an exception, the exception is recorded in the ROB
  - This is why reorder buffers in all new processors

49

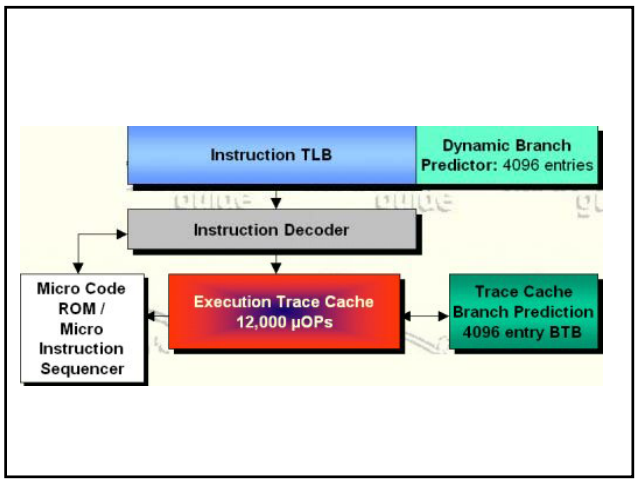
A Quick Case Study—Intel Pentium 4

50

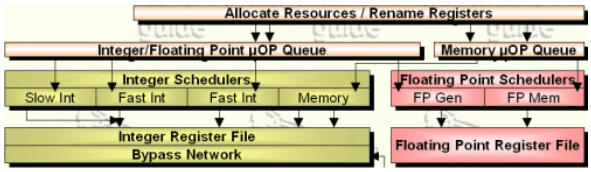
Pentium 4 Architecture

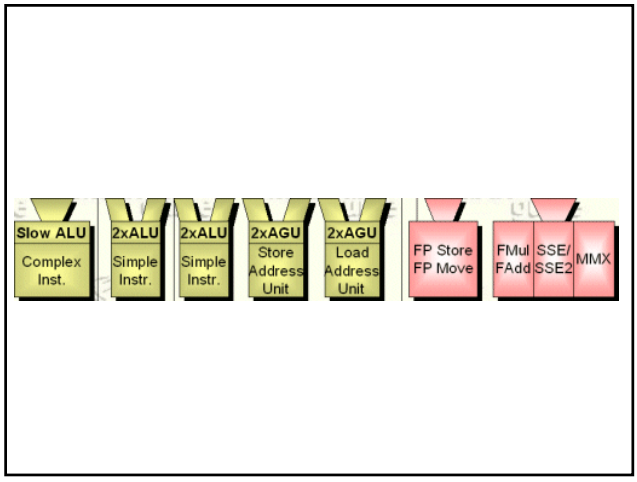


Pentium 4 Pipeline

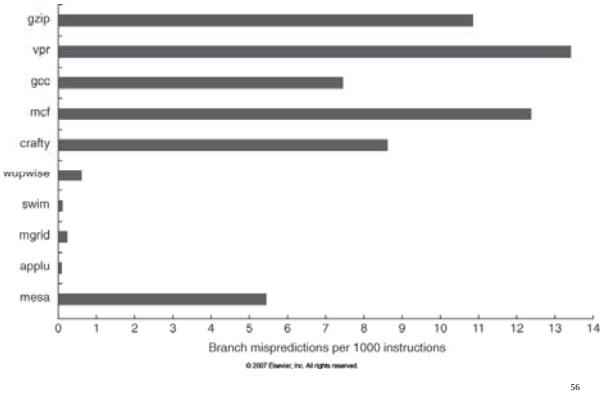


Pentium 4 Dispatch/Execution

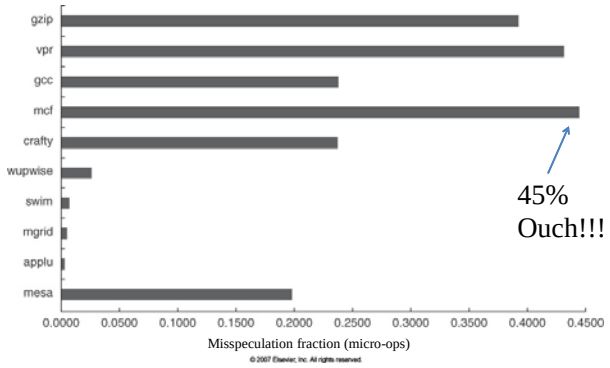




Pentium 4 Branch Prediction Accuracy

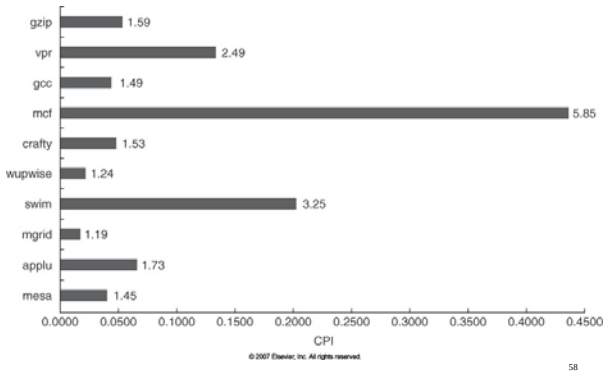


Pentium 4 (Mis)speculation



57

Pentium 4 CPI



58

Perspective

- Goal of multiple-issue architectures is to improve performance without affecting uniprocessor programming model
- Exploiting ILP is conceptually simple, but design problems are very complex in practice
- Processors of last decade (Pentium 4, IBM Power 5, AMD Opteron) have the same basic structure and similar sustained issue rates (3 to 4 instructions per clock) as the 1st dynamically scheduled, multiple-issue processors announced in 1995
- Peak v. delivered performance gap increasing

59

An Attempt to Think Outside the Box: Very Large Instruction Word VLIW)

- Each "instruction" has explicit coding for multiple operations
  - In IA-64, grouping called a "packet"
  - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

60

IA-64 Architecture

- 128 general-purpose registers
- 128 floating-point registers
- Arbitrary number of functional units
- Arbitrary latencies on the functional units
- Arbitrary number of memory ports
- Arbitrary implementation of the memory hierarchy

Needs retargetable compiler and recompilation to achieve maximum program performance on different IA-64 implementations

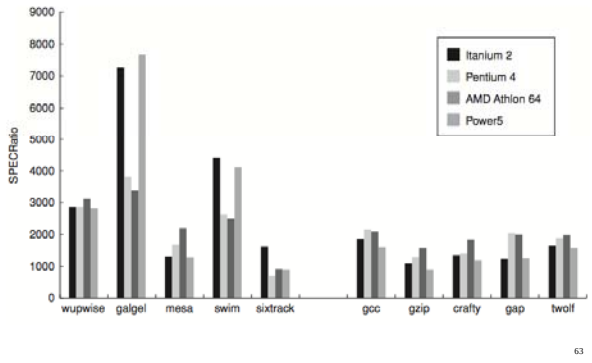
IA-64 Instruction Format

- IA-64 "Bundle"
  - Total of 128 bits
  - Contains three IA-64 instructions *(aka syllables)*
  - Template bits in each bundle specify dependencies both within a bundle as well as between sequential bundles
  - A collection of independent bundles forms a "group

$inst_1$	$inst_2$	$inst_3$	temp

- IA-64 Instruction
  - Fixed-length 40 bits long
  - Contains three 7-bit register specifiers
  - Contains a 6-bit field for specifying one of the 64 one-bit predicate registers

Performance Itanium 2 versus traditional Superscalar



63