UNIT - IV

INSTRUCTION –LEVEL PARALLELISM – 2:

Exploiting ILP using multiple issue and static scheduling

Exploiting ILP using dynamic scheduling

Multiple issue and speculation

Advanced Techniques for instruction delivery and Speculation

The Intel Pentium 4 as example. **7 Hours**

UNIT IV

INSTRUCTION –LEVEL PARALLELISM – 2

What is ILP?

• Instruction Level Parallelism
  – Number of operations (instructions) that can be performed in parallel
• Formally, two instructions are parallel if they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls assuming that the pipeline has sufficient resources
  – Primary techniques used to exploit ILP
• Deep pipelines
• Multiple issue machines
• Basic program blocks tend to have 4-8 instructions between branches
  – Little ILP within these blocks
  – Must find ILP between groups of blocks

Example Instruction Sequences

• Independent instruction sequence:

```
lw $10, 12($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
```

• Dependent instruction sequence:

```
lw $10, 12($1)
sub $11, $2, $10
and $12, $11, $10
or $13, $6, $7
add $14, $8, $13
```

Finding ILP:

• Must deal with groups of basic code blocks
• Common approach: loop-level parallelism

– Example:

– In MIPS (assume $s0 initialized properly):

```
for (i=1000; i > 0; i--)
x[i] = x[i] + s;
Loop: lw $t0, 0($s1) # t0 = array element
addu $t0, $t0, $s2 # add scalar in $s2
sw $t0, 0($s1) # store result
addi $s1, $s1, -4 # decrement pointer
bne $s1, $0, Loop # branch $s1 != 0
```

Loop Unrolling:

• Technique used to help scheduling (and performance)

• Copy the loop body and schedule instructions from different iterations of the loop gether

• MIPS example (from prev. slide):

```
Loop: lw $t0, 0($s1) # t0 = array element
addu $t0, $t0, $s2 # add scalar in $s2
sw $t0, 0($s1) # store result
lw $t1, -4($s1)
addu $t1, $t1, $s2
sw $t1, -4($s1)
addi $s1, $s1, -8 # decrement pointer
bne $s1, $0, Loop # branch $s1 != 0
```

Note the new register & counter adjustment!

• Previous example, we unrolled the loop once

– This gave us a second copy

• Why introduce a new register ($t1)?

– Antidependence (name dependence)

• Loop iterations would reuse register $t0

• No data overlap between loop iterations!

• Compiler *RENAMED* the register to prevent a "dependence"

– Allows for better instruction scheduling and identification of true dependencies

• In general, you can unroll the loop as much as you want

– A factor of the loop counter is generally used

– Limited advantages to unrolling more than a few times

Loop Unrolling: Performance:

• Performance (dis)advantage of unrolling

– Assume basic 5-stage pipeline

• Recall lw requires a bubble if value used immediately after

• For original loop

– 10 cycles to execute first iteration

– 16 cycles to execute two iterations
• Assuming perfect prediction
• For unrolled loop
– 14 cycles to execute first iteration -- without reordering
• Gain from skipping addi, bne
– 12 cycles to execute first iteration -- with reordering
• Put lw together, avoid bubbles after ea

Loop Unrolling: Limitations

• Overhead amortization decreases as loop is unrolled more
• Increase in code size
– Could be bad if ICache miss rate increases
• Register pressure
– Run out of registers that can be used in renaming process
–

Exploiting ILP: Deep Pipelines

Deep Pipelines
• Increase pipeline depth beyond 5 stages
– Generally allows for higher clock rates
– UltraSparc III -- 14 stages
– Pentium III -- 12 stages
– Pentium IV -- 22 stages
• Some versions have almost 30 stages
– Core 2 Duo -- 14 stages
– AMD Athlon -- 9 stages
– AMD Opteron -- 12 stages
– Motorola G4e -- 7 stages
– IBM PowerPC 970 (G5) -- 14 stages
• Increases the number of instructions executing at the same time
• Most of the CPUs listed above also issue multiple instructions per cycle

Issues with Deep Pipelines

• Branch (Mis-)prediction
– Speculation: Guess the outcome of an instruction to remove it as a dependence to other instructions
– Tens to hundreds of instructions "in flight"
– Have to flush some/all if a branch is mispredicted
• Memory latencies/configurations
– To keep latencies reasonable at high clock rates, need fast caches
– Generally smaller caches are faster
– Smaller caches have lower hit rates
• Techniques like way prediction and prefetching can help lower latencies

Optimal Pipelining Depths

• Several papers published on this topic
– Esp. the 29th International Symposium on Computer Architecture (ISCA)

– Intel had one pushing the depth to 50 stages
– Others have shown ranges between 15 and 40
– Most of the variation is due to the intended workload

Exploiting ILP: Multiple Issue Computers

Multiple Issue Computers

• **Benefit**
– CPIs go below one, use IPC instead (instructions/cycle)
– Example: Issue width = 3 instructions, Clock = 3GHz
• Peak rate: 9 billion instructions/second, IPC = 3
• For our 5 stage pipeline, 15 instructions "in flight" at any given time
• Multiple Issue types
– Static
• Most instruction scheduling is done by the compiler
– Dynamic (superscalar)
• CPU makes most of the scheduling decisions
• Challenge: overcoming instruction dependencies
– Increased latency for loads
– Control hazards become worse
• Requires a more ambitious design
– Compiler techniques for scheduling
– Complex instruction decoding logic

Exploiting ILP:Multiple Issue Computers Static Scheduling

Instruction Issuing
• Have to decide which instruction types can issue in a cycle
– Issue packet: instructions issued in a single clock cycle
– Issue slot: portion of an issue packet
• Compiler assumes a large responsibility for hazard checking, scheduling, etc.

Static Multiple Issue

For now, assume a "souped-up" 5-stage MIPS pipeline that can issue a packet with:
– One slot is an ALU or branch instruction
One slot is a load/store instruction

Instruction Type	Pipeline Stages						
ALU or Branch instruction	IF	ID	EX	MEM	WB		
Load or Store instruction	IF	ID	EX	MEM	WB		
ALU or Branch instruction		IF	ID	EX	MEM	WB	
Load or Store instruction		IF	ID	EX	MEM	WB	
ALU or Branch instruction			IF	ID	EX	MEM	WB
Load or Store instruction			IF	ID	EX	MEM	WB

–

Static Multiple Issue

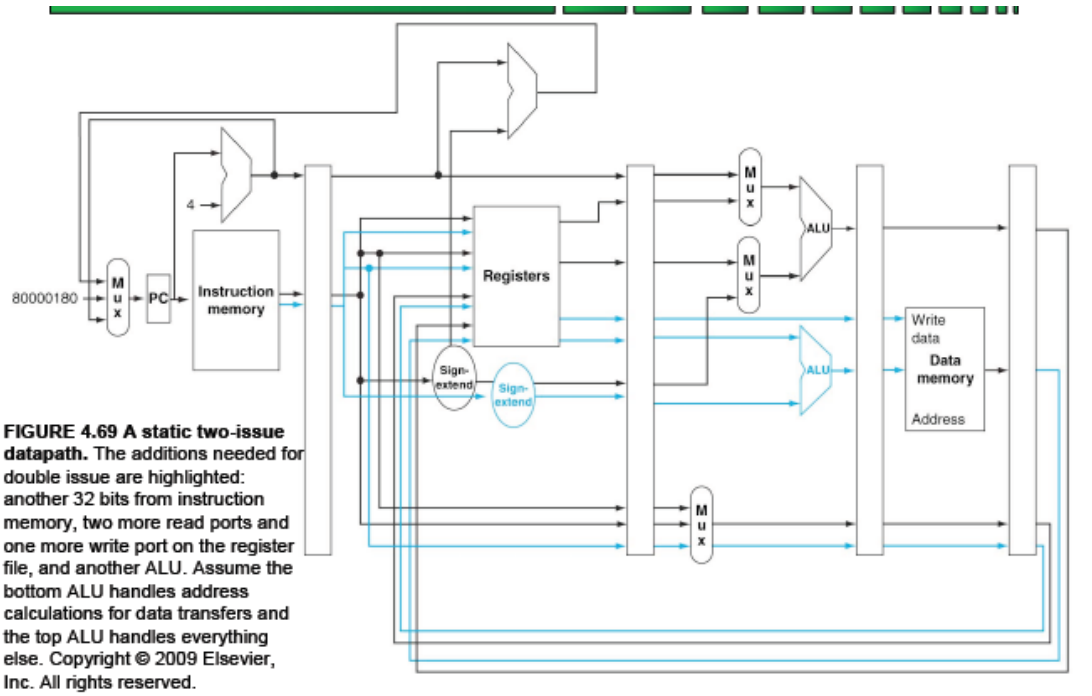


FIGURE 4.69 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else. Copyright © 2009 Elsevier, Inc. All rights reserved.

Static Multiple Issue Scheduling

```
add  $1, $2, $3
add  $5, $2, $2
load $4, $3(100)
load $3, $2(100)
sub  $2, $5, $3
add $2, $2, $4
```

Becomes

Cycle	ALU/Branch Instruction	Load/Store Instruction
1		
2		
3		
4		
5		
6		

Static Mult. Issue w/Loop Unrolling

Original loop schedule for a 2-issue MIPS

```
Loop: lw    $t0, 0($s1)
      addu  $t0, $t0, $s2
      sw    $t0, 0($s1)
      addi  $s1, $s1, -4
      bne   $s1, $0, Loop
```

	ALU/Branch	Load/Store	Cycle
Loop:			1
			2
			3
			4

Unrolled (once) loop schedule for a 2-issue MIPS

```
Loop: lw    $t0, 0($s1)
      addu  $t0, $t0, $s2
      sw    $t0, 0($s1)
      lw    $t1, -4($s1)
      addu  $t1, $t1, $s2
      sw    $t1, -4($s1)
      addi  $s1, $s1, -8
      bne   $s1, $0, Loop
```

	ALU/Branch	Load/Store	Cycle
Loop:			1
			2
			3
			4
			5
			6
			7

Static Mult. Issue w/Loop Unrolling

```
Loop: lw    $t0, 0($s1)
      addu  $t0, $t0, $s2
      sw    $t0, 0($s1)
      addi  $s1, $s1, -4
      bne   $s1, $0, Loop
```

```
Loop:
```

Unroll loop 3x, (4 iterations total) and schedule for a 2-issue MIPS

	ALU/Branch	Load/Store	Cycle
Loop:			1
			2
			3
			4
			5
			6
			7
			8
			9
			10
			11
			12

Exploiting ILP:Multiple Issue Computers Dynamic Scheduling

Dynamic Multiple Issue Computers

• Superscalar computers
• CPU generally manages instruction issuing and ordering
  – Compiler helps, but CPU dominates
• Process
  – Instructions issue in-order
  – Instructions can execute out-of-order
• Execute once all operands are ready
  – Instructions commit in-order
• Commit refers to when the architectural register file is updated (current completed state of program

Aside: Data Hazard Refresher

• Two instructions (i and j), j follows i in program order
• Read after Read (RAR)
• Read after Write (RAW)
  – Type:
  – Problem:
• Write after Read (WAR)
  – Type:

– Problem:

• Write after Write (WAW)

– Type: Problem:

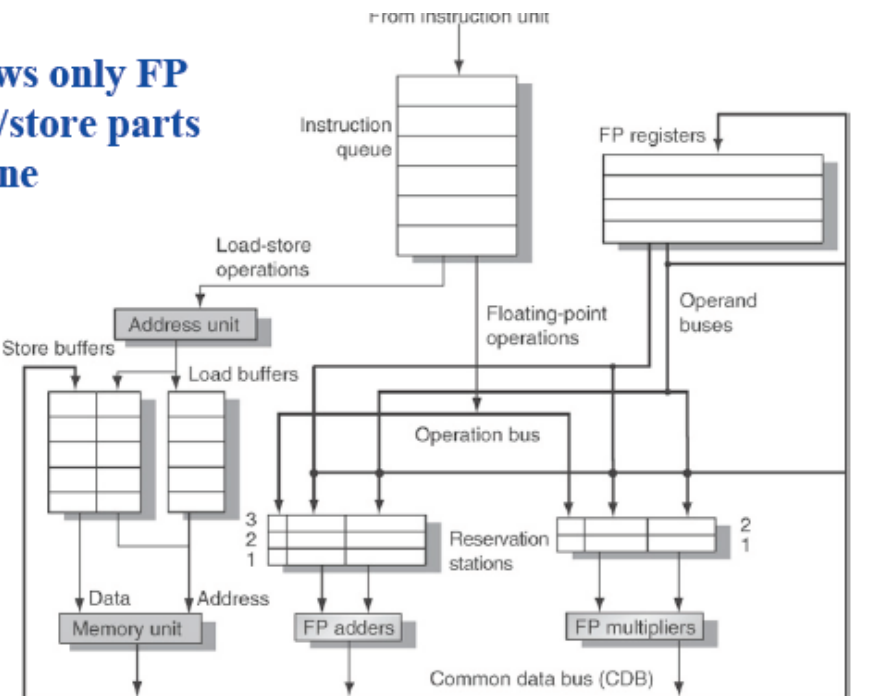
Superscalar Processors

• Register Renaming
– Use more registers than are defined by the architecture
• Architectural registers: defined by ISA
• Physical registers: total registers
– Help with name dependencies
• Antidependence
– Write after Read hazard
• Output dependence
– Write after Write hazard

Tomasulo's Superscalar Computers

• R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM J. of Research and Development*, Jan. 1967
• See also: D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 model 91: Machine philosophy and instruction-handling," *IBM J. of Research and evelopment*, Jan. 1967
• Allows out-of-order execution
• Tracks when operands are available
– Minimizes RAW hazards
• Introduced renaming for WAW and WAR hazards

Tomasulo's Superscalar Computers



Instruction Execution Process

• Three parts, arbitrary number of cycles/part
• Above does not allow for speculative execution
• Issue (aka Dispatch)
– If empty reservation station (RS) that matches instruction, send to RS with operands rom register file and/or know which functional unit will send operand
- If no empty RS, stall until one is available

Rename registers as appropriate
Instruction Execution Process
• Execute
– All branches before instruction must be resolved
• Preserves exception behavior
– When all operands available for an instruction, send it to functional unit
• Monitor common data bus (CDB) to see if result is needed by RS entry
– For non-load/store reservation stations
• If multiple instructions ready, have to pick one to send to functional unit
– For load/store
• Compute address, then place in buffer
• Loads can execute once memory is free
• Stores must wait for value to be stored, then execute

Write Back
– Functional unit places on CDB
• Goes to both register file and reservation stations
– Use of CDB enables forwarding for RAW hazards
– Also introduces a latency between result and use of a value

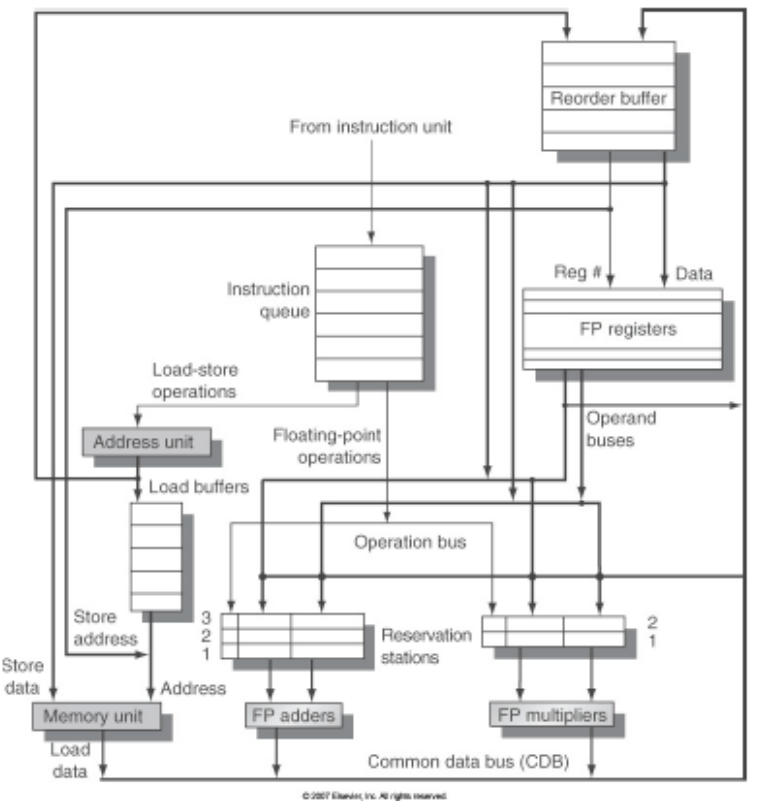
Reservation Stations

• Require 7 fields
– Operation to perform on operands (2 operands)
– Tags showing which RS/Func. Unit will be producing operand (or zero if operand available/unnecessary)
– Two source operand values
– A field for holding memory address calculation data
• Initially, immediate field of instruction
• Later, effective address
– Busy
• Indicates that RS and its functional unit are busy
• Register file support
– Each entry contains a field that identifies which RS/func. unit will be writing into this entry (or blank/zero if noone will be writing to it) Limitation of Current Machine

Instruction execution requires branches to be resolved

• For wide-issue machines, may issue one branch per clock cycle!
• Desire:
– Predict branch direction to get more ILP
– Eliminate control dependencies
• Approach:
– Predict branches, utilize *speculative* instruction execution
  – Requires mechanisms for "fixing" machine when speculation is incorrect
  Tomasulo's w/Hardware Speculation

This shows only FP and load/store parts of machine



Tomasulo's w/HW Speculation

• Key aspects of this design
– Separate forwarding (result bypassing) from actual instruction completion
• Assuming instructions are executing speculatively
• Can pass results to later instructions, but prevents instruction from performing updates that can't be "undone"
– Once instruction is no longer speculative it can update register file/memory
• New step in execution sequence: instruction commit
• Requires instructions to wait until they can commit Commits still happen in order Reorder Buffer (ROB)

Instructions hang out here before committing
• Provides extra registers for RS/RegFile

– Is a source for operands
• Four fields/entry
– Instruction type
• Branch, store, or register operation (ALU & load)
– Destination field
• Register number or store address
– Value field
• Holds value to write to register or data for store
– Ready field
• Has instruction finished executing?
• Note: store buffers from previous version now in ROB

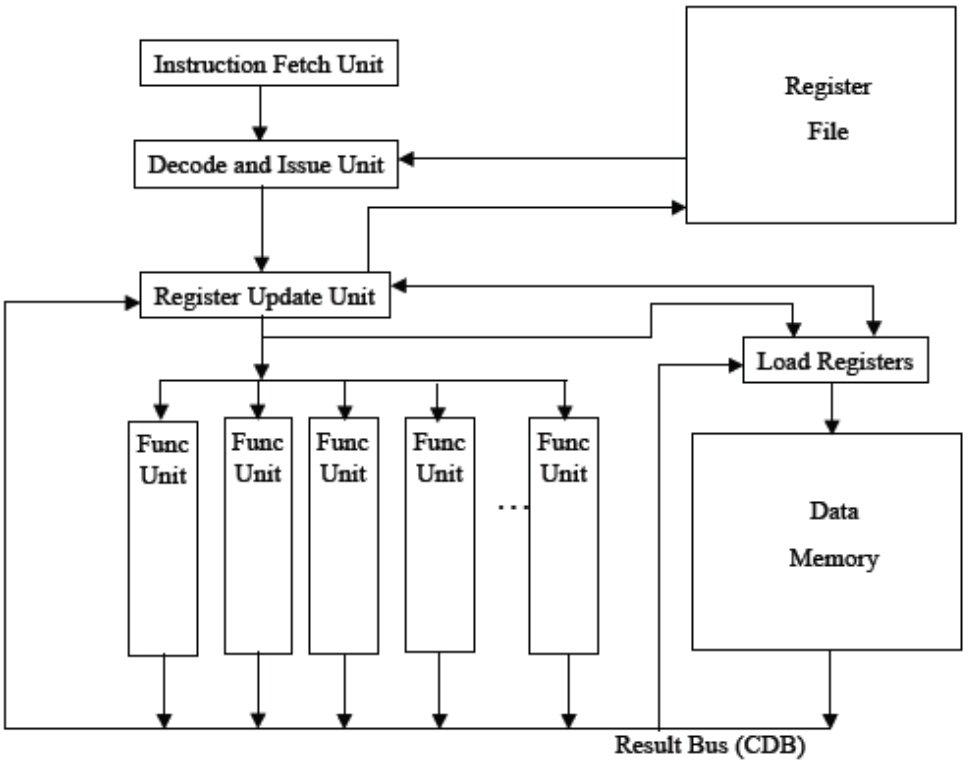
Instruction Execution Sequence

• Issue
– Issue instruction if opening in RS & ROB
– Send operands to RS from RegFile and/or ROB
• Execute
– Essentially the same as before
• Write Result
– Similar to before, but put result into ROB
• Commit (next slide)

Committing Instructions

Look at head of ROB
• Three types of instructions
– Incorrectly predicted branch
• Indicates speculation was wrong
• Flush ROB
• Execution restarts at proper location – Store
• Update memory
• Remove store from ROB
– Everything else
• Update registers
• Remove instruction from ROB

RUU Superscalar Computers



Modeling tool Simple Scalar implements an RUU style processor

– You will be using this tool after Spring Break

• Architecture similar to speculative Tomasulo's

• Register Update Unit (RUU)

– Controls instructions scheduling and dispatching to functional units

– Stores intermediate source values for instructions

– Ensures instruction commit occurs in order!

– Needs to be of appropriate size

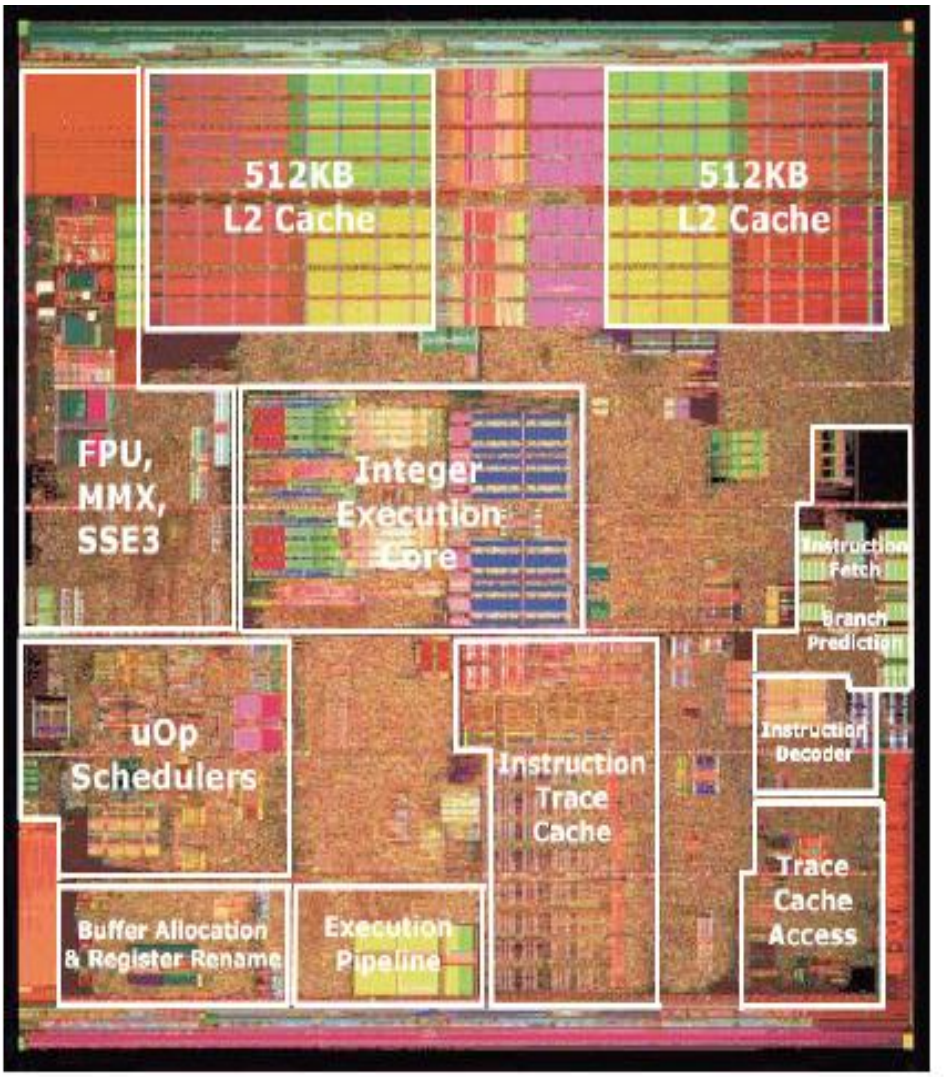
• Minimum of issue width * number of pipeline stages

• Too small of an RUU can be a structural hazard!

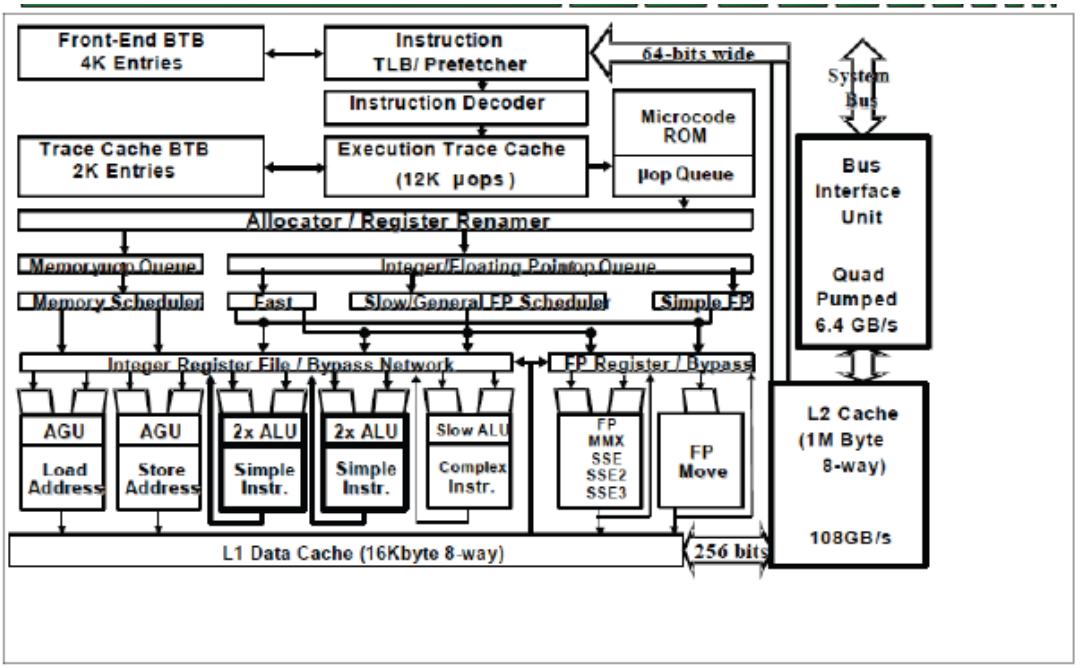
• Result bus could be a structural hazard

A Real Computer:Intel Pentium 4

Pentium 4 Die Photo



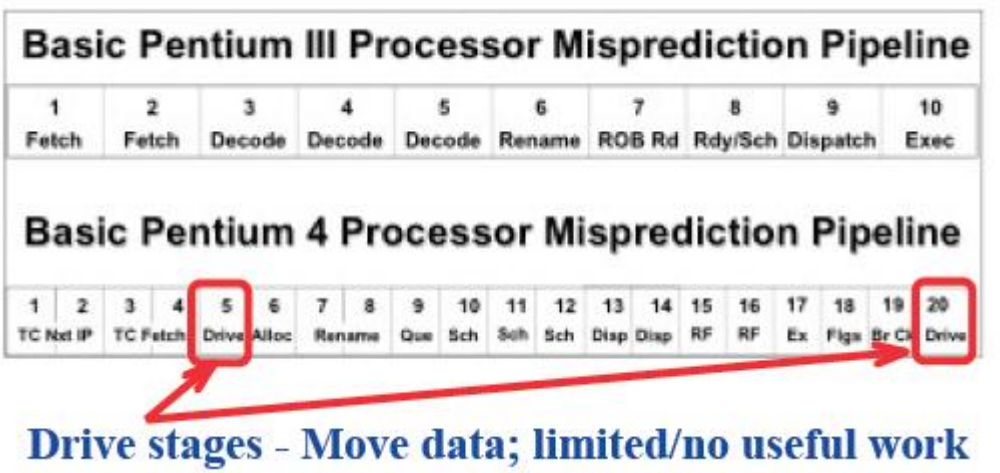
Overview of P4



Boggs et al, "The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology," Intel Tech. J, Vol. 8, Num. 1, 2004

Pentium 4 Pipeline

• See handout for overview of major steps
• Prescott (90nm version of P4) had 31 pipeline stages
  – Not sure how pipeline is divided up
  –



P4: Trace Cache

Non-traditional instruction cache
• Recall x86 ISA

– CISC/VLIW: ugly assembly instructions of varying lengths
– Hard for HW to decode
– Ended up translating code into RISC-like microoperations to execute
• Trace Cache holds sequences of RISC-like micro-ops
– Less time decoding, more time executing
– Sequence storage similar to "normal" instruction cache

P4: Branch Handling

BTBs (Branch Target Buffers)
– Keep both branch history and branch target addresses
• Target address is instruction immediately after branch
– Predict if no entry in BTB for branch
• Static prediction
• If a backwards branch, see how far target is from current; if within a threshold, predict taken, else predict not taken
• If a forward branch, predict not taken
• Also some other rules
• Front-end BTB is L2 (like) for the trace cache BTB (L1 like)

P4: Execution Core

• Tomasulo's algorithm-like
• Can have up to 126 instructions in-flight
– Max of 3 micro-ops sent to core/cycle
– Max of 48 loads, 32 stores
• Send up to 6 instructions to functional units per cycle via 4 ports
– Port 0: Shared between first fast ALU and FP/Media move scheduler
– Port 1: Shared between second fast ALU and Complex integer and FP/Media scheduler
– Port 2: Load
Port 3: Store

P4: Rapid Execution Engine

Execute 6 micro-ops/cycle
– Simple ALUs run at 2x machine clock rate
– Can generate 4 simple ALU results/cycle
– Do one load and one store per cycle
• Loads involve data speculation
– Assume that most loads hit L1 and Data Translation Look-aside Buffer (DTLB)
– Get data into execution, while doing address check
• Fix if L1 miss occurred

P4: Memory Tricks

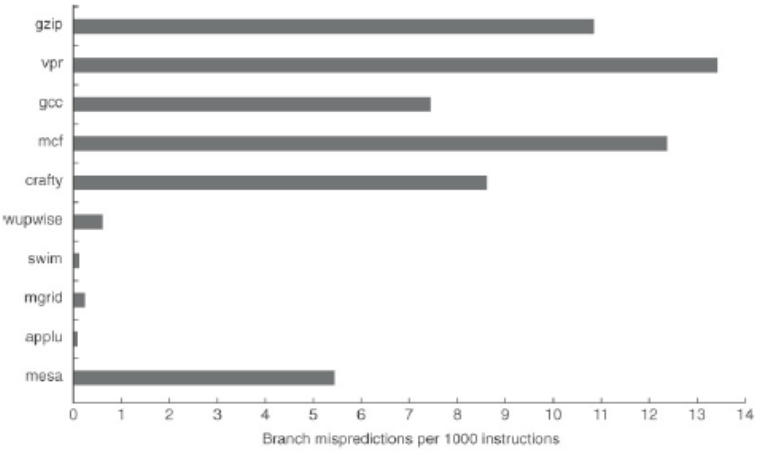
• Store-to-Load Forwarding
– Stores must wait to write until non-speculative
– Loads occasionally want data from store location
– Check both cache and Store Forwarding Buffer
• SFB is where stores are waiting to be written
– If hit when comparing load address to SFB address, use SFB data, not cache data
• Done on a partial address
• Memory Ordering Buffer
– Ensures that store-to-load forwarding was correct
• If not, must re-execute load
– Force forwarding
• Mechanism for forwarding in case addresses are misaligned
• MOB can tell SFB to forward or not
– False forwarding
• Fixes partial address match between load and SFB

P4: Specs for Rest of Slides

• For one running at 3.2 GHz
– From grad arch book
• L1 Cache
– Int: Load to use - 4 cycles
– FP: Load to use - 12 cycles
– Can handle up to 8 outstanding load misses
• L2 Cache (2 MB)
18 cycle access time

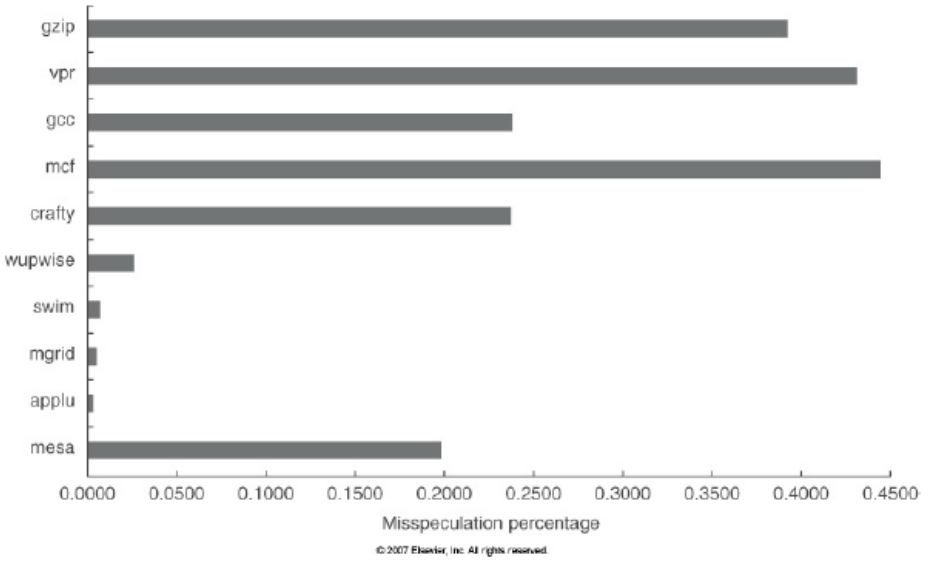
P4: Branch Prediction

- Graph results from subset of SPEC 2000 Benchmarks
  - Integer: gzip, vpr, gcc, mcf, crafty
    - 168 branches/1000 instructions
  - FP: wupwise, swim, mgrid, applu, mesa
    - 48 branches/1000 instructions



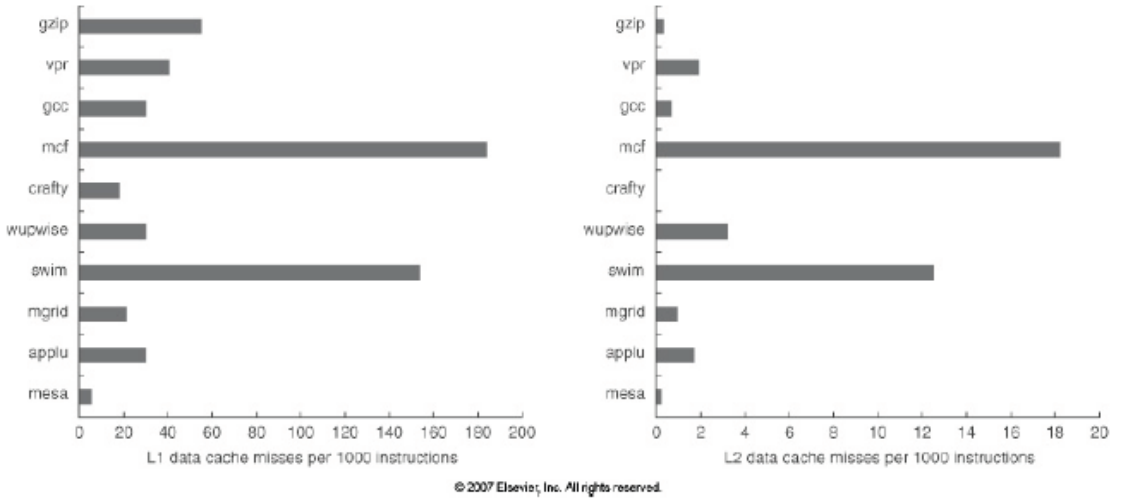
P4: Misspeculation Percentages

- For micro-ops



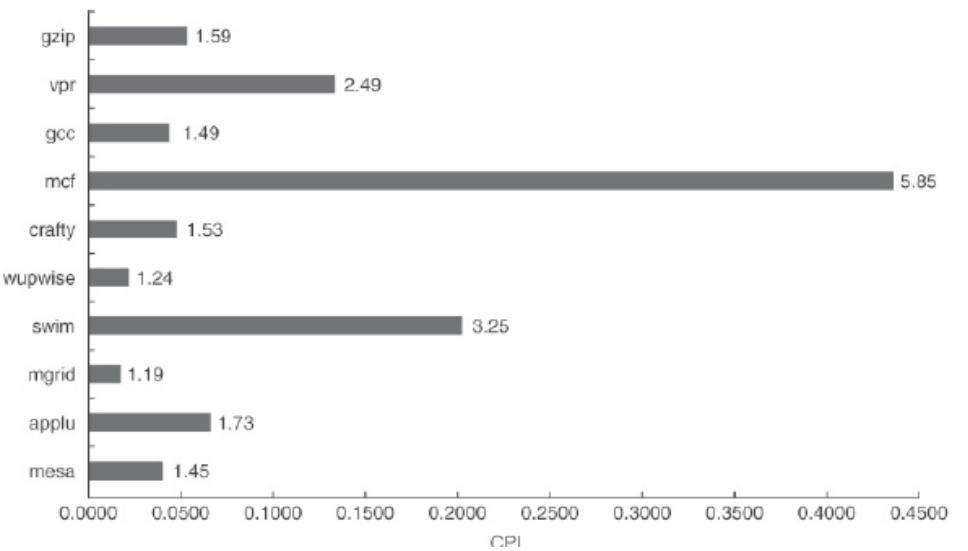
P4: Data Cache Miss Rates

- **This L2 is 2MB, not 1MB (as in paper you read)**
- **Note scale is 10x for L1 as compared to L2**



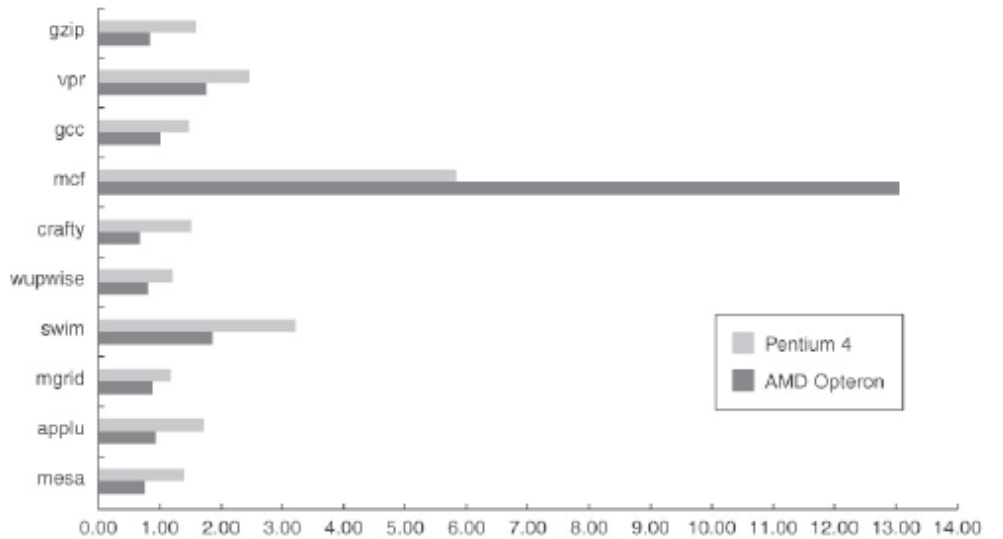
P4: CPI

- **Read values from lines, not sure why X-axis is scaled like it is**
- **1.29 micro-ops per IA-32 instruction**



P4 vs. AMD Opteron

- **Architecturally similar**
  - **Opteron pipeline much shorter (12 stages)**
  - **P4 seems to have a larger cache**
- **CPI comparison below (Opteron at 2.6GHz)**
  - **Opteron CPI lower by factor of 1.27**



P4 vs. Opteron: Real Performance

- **Clock rates (2005 comparison)**
  - **P4: 3.8 GHz**
  - **Opteron: 2.8 GHz**
- **Opteron performance advantage of about 1.08**