Chapter 2: Instruction Level Parallelism

Dynamic instruction scheduling
- Tomasulo’s algorithm

Advanced superscalar processors
- branch prediction, reorder buffer

Case studies
- R10000, K5, Alpha 21264, P6

Compiler techniques
- scheduling, VLIW trace scheduling

Dynamic Scheduling

more parallelism while keeping the illusion of sequential execution
data dependences: RAW
name dependences: WAR and WAW
- removed by Tomasulo’s (first scheme we will study)
control dependences: branches and jumps
- handled by branch prediction (next scheme we will study)
traps, exceptions
- handled by reorder buffer
- finally, we combine all above schemes

Dynamic Scheduling

code fragment
- divf f0, f2, f4
- addf f10, f0, f8
- multf f7, f8, f14

the problem
- addf stalls due to RAW hazard
- multf stalls because addf stalls (why?)
- EVEN THOUGH NO RAW/WAR/WAW

in-order execution limits performance

Dynamic Scheduling

solutions - static scheduling or dynamic scheduling
static scheduling (software)
- compiler reorganizes instructions
- simpler hardware
- can use more powerful algorithms
dynamic scheduling (hardware)
- handles dependences unknown at compile time (?)
- hardware reorganizes instructions!
- more complex hardware but code more portable (why?)
Dynamic Scheduling

do I = 1, N
    • C[I] = A[I] + s*B[I]

assembly
    • If f0, A(r1)
    • If f2, B (r1)
    • multf f2,f2,f4 # s in f4
    • addf f2,f2,f0
    • sf C(r1), f2

what would in-order (+ dynamic loop unroll) do? can we better?

Register Renaming

registers are *names* for values
think of registers as *tags* NOT storage locations

IBM 360/91

Fast 360 for scientific code
    • completed in 1967
    • no decimal instructions
    • dynamic scheduling FP unit (Tomasulo’s algorithm)
    • predates caches (complex path to memory)

pipelined rather than multiple functional units
    • adder supports 3 instructions
    • multiplier supports 2 instructions

Dynamic DLX-T

Tomasulo’s algorithm
    • distributed hazard detection and control
    • results are bypassed to functional units
    • common data bus (CDB) for results
    • uses tags to identify data values
    • reservation stations distribute control
    • CDB broadcasts all results
Dynamic DLX-T

extend DLX as example
- assume multiple FUs rather than pipelined
- main difference is register-memory instructions
- i.e., DLX does not have them

Tomasulo’s Algorithm

3 major steps - dispatch, issue, complete

dispatch
- get instruction from queue
- ALU op: check for available reservation station
- load: check for available load buffer
- if available: issue and copy ready regs to RS
- if not: stall due to structural hazard

Tomasulo’s Algorithm

issue
- if all operands are available, begin execution
- if not, monitor CDB for operand

complete
- if CDB available, write result on CDB
- if not, stall

Note:
- no checking for WAW or WAR
- CDB broadcasts results to functional units, not just registers
Tomasulo’s Algorithm

Reservation stations
  • handle distributed hazard detection and instruction control
  everything receiving data gets the tag of the data
  • 4-bit tag specifies reservation station or load buffer
    specifies which FU will produce result
  register specifier in instruction is used to assign tags
    • THEN IT IS DISCARDED
    • register specifiers are used ONLY in dispatch

Tomasulo Example

Lf f6, 34(r2)
Lf f2, 45(r3)
multf f0, f2,f4
subf f8, f6, f2
divf f10, f0, f6
addf f6, f8, f2

Tomasulo’s Algorithm

reservation stations
  • op -- opcode
    • Q_j, Q_k -- tag fields (sources)
    • V_j, V_k -- operand values (sources)
    • busy -- currently in use
  register file and store buffer
    • Q_i -- tag field
    • ready -- currently under production

Tomasulo eg. (only first load complete)

Instruction Status (for illustration only, NOT real)
  instruction  dispatch  issue  complete
Lf f6, 34(r2)
Lf f2, 45(r3)
multf f0,f2,f4
subf f8,f6,f2
divf f10,f0,f6
addf f6,f8,f2

Fill register status FIRST in order, THEN reservation stations
### Tomasulo example

<table>
<thead>
<tr>
<th>Reservation stations</th>
<th>name</th>
<th>busy</th>
<th>op</th>
<th>V_j</th>
<th>V_k</th>
<th>Q_j</th>
<th>Q_k</th>
</tr>
</thead>
<tbody>
<tr>
<td>add1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Result status (does renaming - filled in order)

<table>
<thead>
<tr>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th></th>
<th></th>
<th>F30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_i</td>
<td>Ready</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Tomasulo eg. (only mult & div not done)

Instruction Status (for illustration only, NOT real)

<table>
<thead>
<tr>
<th>instruction</th>
<th>dispatch</th>
<th>issue</th>
<th>complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>If f6, 34(r2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If f2, 45(r3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult f0,f2,f4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub f8,f6,f2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>div f10,f0,f6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add f6,f8,f2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NO WAR, so add before div

### Tomasulo example

<table>
<thead>
<tr>
<th>Reservation stations</th>
<th>name</th>
<th>busy</th>
<th>op</th>
<th>V_j</th>
<th>V_k</th>
<th>Q_j</th>
<th>Q_k</th>
</tr>
</thead>
<tbody>
<tr>
<td>add1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Result status (does renaming - filled in order)

<table>
<thead>
<tr>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th></th>
<th></th>
<th>F30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_i</td>
<td>Ready</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Tomasulo Example

loop:

- ld f0, 0(r1)
- mult f4, f0, f2
- sd 0(r1), f4
- sub r1, r1, 8
- bnez r1, loop

Fill register status FIRST in order, THEN reservation stations
Tomasulo eg. (after both loads in execute)

<table>
<thead>
<tr>
<th>Instruction Status (for illustration only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction</td>
</tr>
<tr>
<td>ld f0, 0(r1)</td>
</tr>
<tr>
<td>mult f4, f0, f2</td>
</tr>
<tr>
<td>sd 0(r1), f4</td>
</tr>
<tr>
<td>ld f0, 0(r1)</td>
</tr>
<tr>
<td>mult f4, f0, f2</td>
</tr>
<tr>
<td>sd 0(r1), f4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>load1</th>
<th>load2</th>
<th>load3</th>
<th>store1</th>
<th>store2</th>
<th>store3</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>busy</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Tomasulo example

<table>
<thead>
<tr>
<th>Reservation stations</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>add1</td>
</tr>
<tr>
<td>add2</td>
</tr>
<tr>
<td>add3</td>
</tr>
<tr>
<td>mult1</td>
</tr>
<tr>
<td>mult2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Result status (does renaming - filled in order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ready</th>
</tr>
</thead>
</table>

Tomasulo

CDB is bottleneck
- duplicate
- increases required hardware

complex implementation

Tomasulo

advantages
- distribution of hazard detection, eliminates WAR and WAW

CDB
- broadcasts results to multiple instructions
  - central bottleneck

Register renaming
- eliminates WAR and WAW
- allows dynamic loop unrolling (key if only 4 regs, as in x86
  - requires many associative compares
Summary: Tomasulo

- feature
- structural stall in dispatch for RS
- RAW stall in RS for FU
- WAR from CDB
- WAW copy operand to RS
- WAW register renaming
- logic distributed
- bottleneck one result bus

Superscalar Processors

Limits on pipeline performance
- latch overheads, signal skew
- “atomic” instruction issue logic
- Flynn bottleneck: CPI >= 1 (why?)

how can we make the CPI = 0.5?

Superscalar Processor

IF: parallel access to I-cache, require alignment?

ID: replicate logic, fixed length instrs? hazard checks? dynamic?

EX: parallel/pipelined

MEM: >1 per cycle? If so, hazards, multi-port register, D-cache?

WB: different register files? multi-ported register files?

more things replicated

more possibilities for hazards

more loss due to hazards (why?)
Superscalar Processors

Integer + floating-point
any two instructions
any four instructions
any n instructions?

Elements of Advanced Superscalars

High performance instruction fetching
• multiple instructions per cycle
• branch and jump prediction
dispatch and resolving dependences
• eliminate false (WAR and WAW) dependences
• set up linkages for true dependences (RAW)
• e.g., register renaming
Elements of Advanced Superscalars

Parallel out-of-order instruction issue
Speculative execution
Parallel resources
  • functional units, paths/buses, register ports
high performance memory systems
methods for committing processor state correctly
  • precise interrupts
  • speculative execution

A Generic Superscalar Processor

Instruction Fetching
Multiple instructions must be fetched per cycle
  • => fetch entire cache block (4-8 instrs) from I-cache
if every fourth instr is a branch and you fetch 4 instrs per cycle
  • => every cycle there is a branch fetched
  • but branches will not resolved once per cycle
  • => need good branch prediction

Dynamic Branch Prediction
recap: why? target comp (indep) vs. outcome comp (data dep)
  • have hardware guess whether and where a branch will go
e.g., address instruction
  0x64          bnez r1, +4
  0x74          add r3, r2, r1
start with branch PC (0x64) and produce
  • prediction (T-taken or NT- not taken
  • prediction + target PC (0x74)
**Branch History Table**

Start with branch PC and produce
- prediction (T, NT)

![Diagram of Branch History Table]

2^m entries

**Two-bit counters**

with single prediction bit, two mispredictions on loops
- e.g. T T T N T T T N T T T N T T T N

solution: use saturating counter to implement “hysteresis”

what about special loop predictor?
- may be useful as part of hybrid predictor

**Branch History Table**

Read prediction with LSBs of branch PC
change bit on misprediction
may use bit from wrong PC (aliasing)
better predictors:
two-bit state machine (invented by Jim Smith)
- took prediction accuracy from 0 to ~85%
correlating predictors (invented by Yale Patt)
- took prediction accuracy from ~85% to ~95%
- used by all high-performance microprocessors

**Two-bit counters**

![Diagram of Two-bit Counters]
Correlating Predictors

different branches may be correlated:
if (aa == 2) aa = 0;
if (bb == 2) bb = 0;
if (aa != bb) { . . .
if the first two are true, third is false
save recent branch outcomes (approximates the path thru code)
• use branch history register (BHR) to hold recent outcomes
• use both PC and BHR to access table
• recent proposals keep more complete path information

Multi-level Predictors (Yeh and Patt)

PC and BHR can be combined as:
• concatenated
• completely overlapped
• partially overlapped
Correlation can be with itself (local) or with other branches (global)
• accordingly - 4 kinds of predictors: GAg, PAg, PAp, and GAp (same as GShare)

Gshare predictor (McFarling)

Branch Prediction
2^m 2-bit counters

Global Pattern History Table

Global BHR

Program Counter

m bits

Branch History Table
PAg - Local Correlation (with itself)

Local or Global?

A given branch prefers local or global correlation
  • e.g., loop branches prefer local (don’t correlate with other branches in the loop body)

Real processors use two predictors one local and one global and for every branch we choose between the two predictors based on whether the branch prefers local or global
  • we know the preference based on whichever predictor is correct more often for a given branch
  • use a third basic 2-bit predictor to track the preference
  • if local is correct often then the 2-bit counter saturates to local, and if global is correct often then saturate to global

PAp (local correlate + avoid conflict in table)

Branch Target Buffer

already covered

Start with branch PC and produce target PC
Target PC available 1 cycle earlier
  • no bubble for correct predictions

E.g. implementation: a special cache
  • index and tags: branch PC, data: target PC
  • predicted outcome may be held in same table
Branch Target Buffer

considerations
• many more bits per entry than branch prediction table
• size and associativity
• can be associated with the I-cache
• store not-taken branches?

Jump Prediction

subroutine call/return
• use call-return stack (in hardware, invisible to software)
• this is NOT the same as the program stack

general indirect jumps
• more than one target => not just T/NT
• path-based schemes may have potential (later?)

Dynamic Branch Prediction Summary

• results eventually always correct
• squash mispredicted instructions
• don’t slow clock cycle
• very fast on correct predictions
• high prediction accuracy
• not too slow for incorrect predictions

bottomline
• useful for single-issue pipes, critical for multiple-issue pipes
modern CPUs use 2 predictors & choose via a 2-bit predictor

Traps and Interrupts

If f6, 34(r2)
multf f0,f2,f4
subf f8,f6,f2
divf f10,f0,f6
addf f6,f8,f2

what happens if divf causes an exception (addf already done)
what if the first If gets a page fault (multf and addf may be done)
out-of-order completion makes interrupts REALLY hard
Interrupts and mispredictions

Both require similar action:
- offending could be mispredicted branch or excepting instr
- all instrs after should look like they are not done
- all instrs before should be done
- continue from offending instruction

So similar solutions should work

Possible Solutions for Interrupts

Imprecise interrupts - ignore the problem!
- makes page faults (any restartable exceptions) hard
- IEEE standard strongly suggests precise interrupts

In-order completion - stall pipe if necessary

Software clean-up - save info for trap handlers
- implementation dependent

Hardware clean-up - restore to consistent state
- re-order instructions at commit
- complete out-of-order, commit in-order

Model Architecture (paper + book)

Result shift register (RSR)
- control result bus - at most one completion per cycle

<table>
<thead>
<tr>
<th>stage</th>
<th>FU</th>
<th>Dest</th>
<th>Valid</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>fp</td>
<td>4</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Jim Smith and Andy Pleszkun

- used by all high-performance microprocessors

An instruction taking i cycles reserves stage i when it issues
- stalls if stage i is already taken (tries for i+1)

Shift down each cycle

Instruction in stage 1 gets result bus
In-order completion

Reserve preceding stages
• an instrn taking j stages reserves all available i<j stages
• prevents later (sequentially) instr from completing before it
must synchronize stores
• wait for RSR to empty
• put dummy store in RSR

what if store causes exception?

Re-order Buffer

• instructions complete out-of-order
• re-order buffer reorganizes instructions
• modify state (commit) in-order
• add tag to RSR to point to corresponding ROB entry

<table>
<thead>
<tr>
<th>entry</th>
<th>dest</th>
<th>result</th>
<th>excep</th>
<th>valid</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Re-order Buffer

multiple bypass paths?
• => implement in a register file with head/tail pointers

on dispatch, allocate ROB entry and RSR entry
• make RSR point to ROB

on completion, put result in ROB entry pointed to by RSR tag
• mark ROB entry valid

when head of ROB is valid, commit
• update register and exceptions

hardware
• need bypass paths from ROB (PLUS normal bypass)
• implies many ports to the ROB
History Buffer

instead of reorder buffer
  • instructions complete out-of-order
  • register file updated immediately
    • later consumers get values without ROB bypass
    • PLUS normal bypass for immediate consumers
  • hardware cleans-up on exception

History buffer (HB)
  • logs previous value of register (previous in program order)
  • used to undo instructions

<table>
<thead>
<tr>
<th>entry</th>
<th>dest</th>
<th>result</th>
<th>excep</th>
<th>valid</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>408000</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>42</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Histroy Buffer

on issue, copy previous value of dest reg to HB
  • previous value in program order
on completion, mark HB entry valid
on exception, mark exception bit
when head of HB contains an entry marked valid
  • reuse entry (increment head)

when an entry has exception
stall issue, wait for pipe to empty
roll back register values from tail to head
PC at head is precise PC
hardware
  • 3 read ports
  • bypassing is slightly more complex (what?) - IN ADDITION to normal bypass
Future File

- separate register file
- architectural file updated in-order
- future file updated out-of-order
- re-order buffer controls updates to architecture file

Future File

future file is managed like an ordinary imprecise pipeline
architecture file managed like re-order buffer scheme
  - when head entry valid
  - on exception
     - architecture file contains precise state

advantages
  - state saving is easy
  - no extra bypassing as needed by history buffer (but there is normal bypass)

Performance Comparison

simulations using livermore loops
compare three options
  - in-order completion
  - reorder buffer
  - reorder buffer with bypass
    - same as future file and history file
examine both ways of handling stores
  - blocking (in-order) vs non-blocking (out-of-order, store buffer)
relative performance: \( \frac{\text{Time}_{\text{precise interrupts}}}{\text{Time}_{\text{imprecise interrupts}}} \)
Precise Interrupts Summary

RSR is not the only way to control the result bus
RSR will work only if timing is known apriori (caches?)
could have bus arbitration to decide who gets the bus
arbiter will be requested as and when needed and not apriori
the key for precise state is reorder buffer (or the history buffer)
  • complete out of order (overlap execution of instrs in buffer)
  • commit in order
  • consider interrupts of ONLY instruction at commit point
  • if committing instr interrupts, squash all later instructions

Dispatch and Issue

we return to superscalars: generic implementation

Design options for
registrers
  • may be logical == physical; rename in ROB
  • logical != physical i.e., rename in physical
reservation stations
  • may be queues (FIFO) or random access
  • may be unified or partitioned by units

In the following slides, we are going to discuss design options for
  • dispatch and issue of dynamically scheduled superscalars
  • case studies of K5, P6
Reservation stations

- a) Single, shared queue
- b) Multiple queues; one per instruction type
- c) Multiple reservation stations; one per instruction type

Tomasulo’s (old implementation imprecise)

- Logical registers == physical registers
- Reservation stations are randomly accessed
- No reorder buffer => imprecise interrupts

POWER1-RS6000 method (also imprecise)

- Instruction queues, not reservation stations
  - Dynamic issue between queues, not within a queue
- Conventional dataflow- only between registers and functional units
- Registers are renamed
- Precise state maintained in piece-meal fashion
  - Condition registers use history file
  - Integer instrs complete in order (FP not allowed to pass int)
  - Floating point exceptions are imprecise
Renaming - R10000 (precise method 1)

more physical registers than logical registers
  - Tomasulo’s tags == phys. register numbers (you will see)
renaming avoids WAW and WAR hazards
turns instruction stream into “single assignment” form
  - VERY important to get things right (you will see)
assume 8 logical, 16 physical registers

R10000 Renaming Example

register mapping before: (logical physical pairs)
  - L0 P12
  - L1 P13
  - L2 P15
  - L3 P14
  - L4 P9
  - L5 P7
  - L6 P6
  - L7 P8 (free pool: P0, P1, P2, P3, P4, P5, P10, P11)

R10000 Renaming

If f6, 34(r20
If f2, 45(r3)
multf f0, f2, f4
subf f7, f6, f4
divf f1, f0, f6
addf f6, f7, f2

R10000 Renaming Example

register mapping after: (logical physical pairs)
  - L0 P2
  - L1 P4
  - L2 P1
  - L3 P14
  - L4 P9
  - L5 P7
  - L6 P5 (first L6 -> P0 for if f6, and then L6 -> P5 for addf f6)
  - L7 P3 (free pool: P10, P11)
R10000 Renaming Example

code with physical register assignments

If p0, 34(r2)
If p1, 45(r3)
multf p2, p1, p9
subf p3, p0, p1
divf p4, p2, p0
addf p5, p3, p1

mapping overwritten as we fetch, what happens on squash (due to interrupts or mispredictions)? in a few slides

R10000 Renaming

data movement only between regs and units

• only reg file multiported (4-wide issue => 8 rd/4 wr ports)
• ROB does not hold values => not multiported
• associative search in reservations stations
  • for bypassing to immediate consumers
  • as original Tomasulo, unavoidable in out-of-order
  • irrespective of rename scheme: phys reg or ROB
  • fundamental limit on clock speed (you will see)

reorder buffer used for control
  • register reservations + freeing physical regs to free pool

R10000 Renaming

register reservation bits

• apply to physical registers
  • monitored by instructions in RS (for cases where producer completes WHILE consumer is in reservation station)

physical register numbers ARE the tags used by renaming

• single assignment so same tag cannot be written twice
• so consumers cannot get confused
MIPS R10000 Renaming

reorder buffer (ROB) contents:
  • PC, errors, result register, previous mapping (NOT value)
  • PREVIOUS physical register mapping is copied to ROB
    • when a new mapping is made
  • similar to history buffer approach
    • except there we copied previous value
    • here we copy previous physical mapping
    • (mapping and value are equivalent)

Example of R10000 Speculative Execution

lf r6, 34(r2)
lf r5, 38(r2)
add r0, r5, r6
add r2, r2, r1
bnez r0, -16
lf r6, 34(r2)
lf r5, 38(r2)
add r0, r5, r6

R10000 Speculation (interrupts & br pred)

predict branches and speculatively execute instructions

correct: great!

incorrect: squash later speculative instrs (sounds familiar?)

reorder buffer can be used for squashing
  • add “speculative bit” to reorder buffer
  • do not commit any instruction if speculative bit is set
  • correct => clear speculative bit
  • incorrect => squash speculative instrs in reorder buffer
  • use “old mapping information” to restore previous mapping

Example of R10000 Speculative Execution

• register mapping at time of speculative branch:
  • logical physical map (free pool at start: 0,2,3,4,5,6,8,15)
    • L0 P3
    • L1 P7
    • L2 P4
    • L3 P13
    • L4 P14
    • L5 P2
    • L6 P0
    • L7 P12
Example of R10000 Speculative Execution

**register mapping when branch is decided**

- logical physical map
  - L0  P8
  - L1  P7
  - L2  P4
  - L3  P13
  - L4  P14
  - L5  P6
  - L6  P5
  - L7  P12

---

Example of R10000 Speculative Execution

**History Buffer**

<table>
<thead>
<tr>
<th>entry</th>
<th>instr</th>
<th>current map</th>
<th>old map</th>
<th>speculative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ld</td>
<td>P0</td>
<td>P11</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>ld</td>
<td>P2</td>
<td>P10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>add</td>
<td>P3</td>
<td>P9</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>P4</td>
<td>P1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>bnez</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ld</td>
<td>P5</td>
<td>P0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>ld</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>add</td>
<td>P8</td>
<td>P3</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

R10000 freeing physical register

Freeing physical registers (returning to free pool)
- ignoring interrupts,
  - freed after last read is done
  - last read can be detected in many ways
  - usage counters (ugly!)
- considering interrupts (see next slide)
  - freed at logical-overwriting instr commit (clean!)
  - at decode is imprecise, at commit is precise

---

Freeing physical registers with interrupts

lf1 f6, 34(r2)  assume f6 --> p0
lf2 f2, 45(r3)
multf f0, f2, f4
subf f7, f6, f4
divf f1, f0, f6
addf f6, f7, f2

we release p0 as soon as addf is decoded and subf and divf are done (and p0 is reused and overwritten). But LATER lf2 excepts and we squash mul,sub,div, add. When we return from exception, sub and div cannot find f6 value because p0 overwritten
Freeing physical registers with interrupts

if you look at the code you will think releasing p0 when we did is fine

But not so due to an earlier instr excepting later in which case I have to RE-execute previously-done instrs which read p0 AGAIN

Same problem will occur due to branch mispredictions -> replace If f6 with a mispredicted branch

The solution is to free p0 only when I have a guarantee that I will NOT read p0 any more - DESPITE interrupts and mispredictions

That guarantee is when the over-writing instr (addf f6) COMMITs

Renaming and History Buffer summary

Tomasulo’s tags == physical registers

Two uses for the “old mapping info” in History Buffer
  • if instruction commits: free old map
  • if instruction squashes: use old map to restore rename table (i.e., the logical to physical mapping table)

Values are updated imprecisely in the physical reg file
  • but as long as rename maps are precise it is ok

rename maps are updated imprecisely (as instrs are fetched)
  • use history buffer to hold old map & restore precise state

Renaming-PowerPC604 (precise method 2)

logical registers == physical registers
reorder buffer (ROB) holds renamed logical registers
  • need multiporting of ROB (4-wide issue => 8 rd/4 wr ports)
  • AND reg file is multiported (values come from/go to reg file)
reservation stations random access (as usual, search to bypass)
  • as in method 1, fundamental limit on clock speed (more later)

registers read only at dispatch
otherwise values come from functional units or reorder buffer

Sohi’s Method (used in PowerPC 604)
Sohi's Method Example

If f6, 34(r2)
If f2, 45(r3)
multf f0,f2,f4
subf f7, f6, f2
divf f1, f0, f6
addf f6, f7, f2

reorder buffer while FP divide in progress:
entry content
0   f6
1   f2
2   f0
3   f7
4   f1
5   f6

Sohi's Method Example

Tomasulo's tags == ROB entries

Because ROB holds renamed values
commit => reclaim ROB entry => automatically free rename map
no complication of freeing the rename mapping here
  • unlike physical register renaming (as in R10000)
Associative search thru reservation stations

Allows consumers to IMMEDIATELY follow producers

- combined with bypass -- data will come via bypass
- NECESSARY for RAW, independent of renaming method
- more reservation stations => more parallelism
- but in associative search, n entries => n^2 RC delay
- this delay must fit 1 cycle else RAW separated by >1 cycle
- but RAW common and not enough independent instrs to fill >1 cycles
- => this search FUNDAMENTALLY bounds clock speed

Out-of-order loads and stores?

what about RAW, WAR, WAW through memory?

- RAW - previous store, later load to same address
- WAR - previous load, later store to same address
- WAW - previous store, later store to same address

Why different than register hazards?

- registers known in instructions but not memory address
- so existence of hazards known only when ld/st executes
- => if a previous ld or st address not known yet
  - can I execute a later load or store?

Out-of-order loads and stores?

No - stall - allow loads and stores to execute only in order

- then no point in doing out of order for other instrs
- because other instrs are dependent on loads, and
- loads and stores are every 5th instr!

Yes -- lds and sts are recorded in ld/st buffers in program order

- needed to determine hazards through memory
- just like register renaming is done in program order
  - to determine register hazards

Memory Operations
Out-of-order loads and stores?

RAW - load checks PREVIOUS stores in store buffer to see if address match
  • if match then get value from store, not memory
  • if a previous store addr is unknown, go to memory speculatively assuming no match
  • when stores execute they check LATER load addresses
    • if match, squash incorrect load via reorder buffer
    • if not, the speculative load was ok

WAR - stores sent from store buffer to memory in program order previous loads guaranteed to have been done

Out-of-order loads and stores?

WAW - stores go to memory in program order
  • but two stores to same address coexist in store buffers
  • this is a limited form of renaming

Basically what we are saying is loads go when they want but stores go in program order (so stores may be stalled)
  • why this asymmetry?
    • loads overwrite registers - fix with reorder buffer
    • but stores overwrite memory -- harder to fix
  • won’t the stalls decrease performance?

Out-of-order loads and stores?

• No - because lds and sts are asymmetric
• load values immediately needed by later instrs in pipe
  • through register RAW hazard which is common
• store values not immediately needed by pipe
  • by some later load to same address
  • through memory RAW hazard which is less common

Case Study: AMD K5

4-way superscalar - 2 memory, 2 ALU, 1 FP
16 entry reorder buffer
16 KB instruction and 8 KB data cache
CISC instructions converted to ROPs
is the difference between CISC and RISC so important?
Case Study: AMD K5

CISC to ROP Conversion

ROPs are like microcode
predecode helps identify instruction boundaries
CISC instructions placed into Byte Queue
Byte Queue instructions broken into 4 ROPS per cycle

Case Study: Intel P6

uses ROPs like K5
20 unified reservation stations
up to 3 simple instr decode/cycle; only 1 complex
ROB holds values
up to 5 ROPs issue per cycle; 3 ROPS commit per cycle
max 1 load and 1 store per cycle
CPU and L2-cache on MCM
512 entry, 4-way associative BTB
Case Study: Intel P6

12 stage pipeline: good branch prediction
Yale Patt’s PhD student joined Intel to implement it

Canonical OoO Pipeline

F    D    Rename    Dispatch    Issue    RegRd    EX    Mem    WB
F to Disp in predicted program order - Issue to WB out of order
These are “architectural stages”
  • 1 arch stage MAY be 1 or more “circuit stages” (high clock)
  • EXCEPT stages critical for RAW hazards (issue, EX)
In x86, F and D can be many circuit stages (why?)
Reservation stations are collectively called as “issue queue”
Reorder buffer in the background for commit
  • notice that commit is not part of the pipe

Canonical OoO: Back-to-back issue

F    D    Rename    Dispatch    Issue    RegRd    EX    Mem    WB
If we blindly apply Tomasulo’s then tag broadcast will occur after
EX and then consumers will issue
2 stages between issue and EX (more if register file is more than
1 cycle) so RAW hazards separated by 2+ cycles
  • bad because RAW is common
Instead producers do tag broadcast AS SOON AS issued (and
not wait till EX) so consumers can issue back-to-back after
producers (with appropriate time delay, if producer is multi-cycle)
Producers’ values come via bypass JUST IN TIME for consumer
EX
Canonical OoO Pipeline

ROB vs issue queue (ROB size = window size)
- all in-flight instrs in the pipe (between fetch and commit
- ROB - 200+ entries and issue queue - 30-40 entries
- Separate int/FP issue queue (why?) but 1 ROB (why?)

issue q size critical for clock (NOT ROB size -common mistake)
- issue Q is tag-searched for issue and NOT ROB
- completed but not committed instrs wait in ROB - so no search in the ROB
- no need to tag-search completed instrs

Canonical OoO Pipeline: One problem

ld/st instructions can hit or miss in cache - uncertainty in the pipe
So when do instrs dependent on a load issue?
- option1: AFTER ld determines hit or miss in cache
  - 4 arch stages between Issue & Mem
  - lds are common and RAW on load value common
  - many stalls for EVERY lds even if cache hit
- option 2: Predict cache hit (why will that work?)
  - issue dependent instrs after load hit delay
  - load hit delay UNAVOIDABLE (seen this before?)
  - fewer stalls for cache hits (what about cache miss)

Canonical OoO Pipeline: One problem

option 1: dependent instrs 4 stalls (mem/cache access = 2 cycles)
- issue reg ex mem mem
- independent instrs can issue immediately (but?)

option 2: dependent instrs 2 stalls
- how do dependent instrs get load value?
- what if load is a cache miss?
- reissue instrs ONLY between issue & mem (called “replay”)
- not a full blown squash like a br misprediction
  - br mispred squashes ALL later instrs in ROB

Canonical OoO Pipeline

Key clock speed constraints
- Rename, issue, and EX+bypass must be 1 cycle to avoid RAW stalls

Key penalties:
- F to Mem - branch misprediction penalty
- Issue to mem - load miss replay penalty
Superscalar Trends

could continue with wider issue, larger reorder buffer
 increases IPC BUT ignores clock cycle
clock is a function of logic + wire delays
larger windows, larger issue width
  • more associative searches
  • for tag matches in reservation stations
  • => more logic per clock + longer paths per clock

In future, wire delays will dominate (bypass paths critical)
And now, power has stopped this trend

Compiler Scheduling

basic idea
have compiler reorder code to mitigate effect of dependencies
three examples
  • loop unrolling
  • software pipelining
  • trace scheduling

think of the dispatch/issue hardware to move into compiler

Loop Unrolling

with scheduling
loop:
  • ld f0, 0(r1)
  • stall
  • addd f4, f0, f2
  • sub r1, r1, #8
  • bnez r1, loop (delayed branch)
  • sd 8(r1), f4

Loop Unrolling

ld f0, 0(r1)                        ld f10, -16 (r1)
addd f4, f0, f2                    addd f12, f10, f2
sd 0(r1), f4                        sd -16(r1), f12
ld f6, -8(r1)                        ld f14, -24(r1)
addd f8, f6, r2                     addd f16, f14, f2
sd -8(r1), f8                        sub r1, r1, #32
bnez r1, loop (delayed branch)       bnez r1, loop     # delayed branch
sd -24(r1), f16
Loop Unrolling

ld f0, 0(r1)    addd f12, f10, f2
ld f6, -8(r1)   addd f16, f14, f2
ld f10, -16(r1) sd 0(r1), f4
ld f14, -24(r1) sd -8(r1), f4
add f4, f0, f2  sd -16(r1), f12
add f8, f6, f2  sub r1, r1, #32
bnez r1, loop   # delayed branch
    sd -24(r1), f16

Loop Unrolling

overlaps loops if no inter-iteration dependence
will not work for “recurrences”
e.g.,
• for (i=1; i <= N; i++)
  • sum = sum + a[i] * b[i];

Software pipelining will do better
Originally done by microcode people [Charlesworth 1981]
Rediscovered by Monica Lam

Software Pipelining

sum = 0.0;
for (i=1; i <= N; i++)
    sum = sum + a[i] * b[i];

load a[i]
load b[i]
mult ab[i]
add sum[i]
Software Pipelining

for (i=3; i <= N; i++)
    add sum[i-2]
    mult ab[i-1]
    load a[i] # sum = sum + a[i]*b[i]
    load b[i]

finish-up block: mult ab[N];
    add sum[N-1]
    add sum[N]

Software Pipelining: Timing

start-up i=3 - - - - i = N finish-up
a1 a2 +1 +i-2 +N-2
b1 b2 *2 *i-1 *N-1
    *1 a3 ai aN +N-1
    b3 bi bN *N +N

Software Pipelining: Example

start-up block

sum = 0
load r11, 0[rA] # a[1]
load r12, 0[rB] # b[1]
load r21, 4[rA] # a[2]
load r22, 4[rB] # b[2]
mult r13, r11, r12 # ab[1]

loop: add sum, sum, r13 # a+b[i-2]
mult r13, r21, r22 # ab[i-1]
    ld r21, 0[rA]
    ld r22, 0[rB]
    add rA, rA, 4
    add rB, rB, 4
    branch Loop

software-pipelined loop looks inverted vs. original loop
**Software Pipelining: Example**

add sum, sum, r13  # finish up code
mult r13, r21, r22
add sum, sum, r13

**Software Pipelining**

will work for recurrences in loops
will not work for non-loop situations
will not work well for loops with branches

**Trace Scheduling**

• as many as dependence length of recurrence in loop

**Trace Scheduling [Ellis 1985]**

b[i] = "old"
a[i] =
if (a[i] > 0) then
    b[i] = "new";  common case
else
    X
endif
c[i] =

**Trace Scheduling: Top Level Algorithm**

until done
  • select most common path - called a trace
  • schedule trace across basic blocks
  • basic block -
    • code block with single entry point, single exit point
  • repair other paths
Trace Scheduling

trace to be scheduled: repair code
b[i] = "old" Label1: restore old b[i]
a[i] = X
b[i] = "new" recalculate c[i]?
c[i] =
if (a[i] <=0) goto label1
label2:

Static Scheduling: Summary

loop unrolling
+ large block to schedule
+ reduces branch frequency
– expands code size
– have to handle "extra" iterations
software pipelining
+ no dependences in loop body
– does not reduce branch frequency
– need start-up and finish-up blocks

Static Scheduling: Summary

trace scheduling
+ works for non-loops
– more complex than unrolling and software pipelining
– does not seem to handle more general cases

Key Limitations of Static Scheduling

1. ambiguous memory dependences
2. null-pointer dereference and exceptions
3. hard-to-predict branches which break traces

Bob Rau nsolved these issues
• VLIW proponent, created Cydrome multicomputer (1980’s)
• solutions adopted later by Intel IA-64
Ambiguous dependences

software limitation
what compiler can’t analyze, it can’t schedule
ambiguous memory dependences
e.g.,
- \*ptr1 = ---- store instr
- tmp = \*ptr2 ---- load instr
- add (tmp+4) ---- will stall due to dependence on tmp
- but load cannot be moved up: compiler does not know FOR SURE if ptr2 != ptr1 (i.e., ambiguous dependence)

Ambiguous dependences

registers don’t cause similar problems. why?
dependence distance of 5
for (i=0; i<=100; i++)

IA-64’s solution to ambiguous dependence

uses special instrs and hardware: to the eg. code as follows:
- \*ptr2 ---- move load UP & use special ld opcode
- .. some other code ....
- \*ptr1 = ---- store instr
- sentinel (ptr2, repair-code-label) ---- instr to check ptr2
- add (tmp+4) ---- no stall

All stores between special load instr and matching sentinel instr
- put their store addresses in a special buffer
- sentinel instr searches the buffer for match with ptr2

for (i=0; i<=100; i++)
A[a * j + b] = A[c * k + d]
if dependence exists then GCD(c,a) must divide (d-b)
e.g.,
- for (i=0; i<=100; i++)
- a = 2, b = 3, c = 2, d = 0, GCD (a,c) = 2 and d-b = -3.
- so no dependence
works only for array indices but not pointers
**IA-64’s solution to ambiguous dependence**

if no match => the stores were to addresses different than ptr2
if match => load should not have moved up
  • go to repair code specified by repair-code-label

---

**Null pointer dereference**

consider code:

....some code X ...

if (ptr != NULL) {
  # common case ptr != NULL
  tmp = *ptr;
  # load thru ptr
  .. = tmp + 4
  # use load value -- causes stall
}

....some code Y ......

---

**Null ptr (contd)**

form a trace with the IF path and schedule

tmp = *ptr;                 # move the load up
... some code X ....      # get rid of the branch in the trace
.. = tmp + 4;            # use load value without stall
... some code Y ...

if (ptr == NULL) go to repair code

what is wrong?

---

**IA-64’s solution to Null pointer**

if ptr == NULL, the load will cause a segmentation fault!

reg-tmp = *ptr;          # special ld opcode
... some code X ....     # special ld opcode
sentinel (reg-tmp, repair-code)  # checks reg-tmp for seg faults
.. = tmp + 4;            # use load value without stall
... some code Y ......   # use load value without stall
IA-64’s solution to Null pointer

if special load causes segmentation fault
set a “poison bit” on register reg-tmp
sentinel checks the poison bit and goes to repair code

if no seg fault, then sentinel is a no-op

IA-64’s Predication

Originally done in Cray-1 and called vector mask

... if (a < b) then                     # difficult-to-predict branch
        r1 = 4;                                 # short if clause
else
        r1 = 5                                      # short else clause

IA-64’s Predication

predicated code:
pred1 = (a < b);                  # predicate reg holds 1 bit for true/false
(pred-true pred1) r1 = 4         # r1 written if pred1 is true else nop
(pred-false pred1) r1 = 5        # r1 written if pred1 false else nop

uses “conditional” or predicated instructions

IA-64’s Predication

predication removes branches altogether
allows fetch to continue without hinderance
works well if
• if and else clauses are small & no more nested branches
• executes BOTH if and else paths
  + no branch problems
  – guaranteed to utilize only 50% of the machine
  – if another branch, utilization drops to 25% (and 12.5% etc)
  + still better than incorrect branch prediction
Software vs. Hardware

Equivalent techniques, differ in applicability

Hardware
+ high branch prediction accuracy
+ has dynamic information on latencies like cache misses
+ works for generic, non-loop, irregular code
  - e.g., databases, desktop applications, compilers
- limited reorder buffer size - limited "lookahead"
- high cost/complexity

Software vs. Hardware

Software
+ can look at large amounts of code - large "lookahead"
+ no hardware cost
+ works for regular code - "fortran codes"
  - e.g., engineering applications, weather prediction
- low branch prediction accuracy - can improve by profiling
- has no dynamic information on latencies like cache misses
  - run code once to figure branches/cache misses
  - use a different input, not real input

Software vs. Hardware

How did hardware do all our software examples?

Unrolling
  - branch prediction, renaming

Software pipelining
  - prediction + renaming + out-of-order issue

Trace scheduling
  - prediction + renaming + out-of-order issue + squashes

Software vs. Hardware

Sentinel instructions handle
  null pointer -- exception taken at the right point like ROB
  memory disambiguities -- search store addr like ld/st Q

Predication
  avoid hard-to-predict branches
  fetch continues unhindered
  less complex branch predictor maybe?

But what about code from the past - dusty deck codes??
VLIW: All Software

very long instruction word
implement a number of independent functional units
provide a long instruction word with one operation per FU
instruction latencies are fixed
compiler packs independent instructions into VLIW
  • compiler schedules all hardware resources
entire long word issues as a “unit”
result: ILP with simple hardware, simple control, fast clock

Hardware/Software Tradeoffs

Mitigating issues for hardware
compiler can still do higher level scheduling
will hardware control slow clock?
  • Latest x86 Xeon - out of order at 4GHz

hardware scheduling
  + uses runtime info => increased ILP, flexibility
  + complicated hardware
  + limited scope for finding ILP

software scheduling
  • uses only compile-time info
  • simple hardware
  • broader scope of finding ILP