CS/ECE 752 Chapter 3

*Instruction Level Parallelism and its Dynamic Exploitation*

Instructor: Prof. Wood

Computer Sciences Department
University of Wisconsin

Slides developed by Profs. Falsafi, Hill, Smith, Sohi, Vijaykumar, and Wood of Carnegie Mellon University, Purdue University, and University of Wisconsin

---

**Dynamically Scheduled Processors**

- The dynamic scheduling (superscalar) mindset
- Control and Data dependences
- Progression of performance increase
- Multiple instruction issue
- Basic out-of-order execution
- Precise interrupt maintenance
- Speculative execution
- Branch prediction
- Register renaming
- Case studies

---

**The Problem and Mindset**

- Start with a static program
  - program written assuming sequential execution
- Sequence through static program to obtain dynamic sequence of operations
  - operations operate on values (present in storage locations) and create new values
- Develop a *schedule* to execute operations in dynamic sequence
- Operations constitute an *instruction window*
  - instruction window is a portion of the dynamic instruction stream

---

**The Problem and Mindset, contd.**

- *Implement* the execution schedule
- Carry out above at desired speed
  - use overlap techniques to improve speed
- Maintain the appearance of sequential (serial) execution
The Big Picture

Program Form
Static program
Dynamic instruction stream
Execution window
Completed instructions

Processing Phase
Instruction fetch & branch prediction
Dependence checking & dispatch
Instruction issue
Instruction execution
Instruction reorder & commit

Dependences

- Determine the order of operations in an execution schedule
- Two types: control dependences and data dependences

Control Dependences

- True control dependences caused by flow of control through program
- Artificial control dependences caused by sequential nature of writing code: PC determines order in which instructions are to be executed
- Must be overcome to do multiple issue or out-of-order issue

Control Dependence Example

```
for (i = 0; i < last; i++) {
    if (a[i] > a[i + 1]) {
        temp = a[i];
        a[i] = a[i + 1];
        a[i + 1] = temp;
        change++;
    }
}
```
Data Dependences

- Read after Write (RAW): true dependence
- Write after Read (WAR): anti-dependence
- Write after Write (WAW): output-dependence

- Only RAW dependences need to be observed
- WAR and WAW are an artifact of storage name used
  ▫ also called name dependences
  ▫ can be overcome with storage renaming

Converting Control to Data Dependences

- Associate condition operand (guard) with each instruction
- Semantics of instruction: regular semantics if guard is TRUE, NOP if guard is FALSE
- Data dependence between instruction setting guard and instructions using guard
- Called if-conversion
Converting Data to Control Dependences

- Can be done, but not important (yet) for our purpose
- Called reverse if-conversion

Execution Schedules

- Given a set of operations to be executed, along with a dependence relationship, how to develop an “execution schedule”?
- Dependence relationships and operation latencies determine shape of schedule
  - “Better” schedule is one that takes a fewer number of time steps (i.e., is more parallel)
  - Reducing operation latencies allows creation of better schedules
  - Relaxing dependence constraints allows creation of better schedules

Execution Schedule Examples

- Serial schedule
- Pipelined schedule
- Parallel schedule
- Parallel pipelined schedule
- Out-of-order schedule

Progression of Performance Increase

- Serial processing
  - one operation at a time
- Pipelining to overlap operation execution
  - initiate one operation at a time, but overlap processing of multiple operations
- Pipelining with out-of-order execution of instructions
  - allow operations to execute in an order different from which they were initiated
  - target artificial control and data dependences
Progression of Performance Increase

- Parallel pipelines, i.e., multiple instruction issue
  - initiate multiple operations at a time, and overlap processing of multiple operations
  - target artificial control dependences
  - the "original" superscalar
- Parallel pipelines with out-of-order execution of instructions
  - target artificial data and control dependences
- Give the scheduler more choice of operations to choose from
  - use speculation to "overcome true dependences"
  - modern OOO superscalar processor

In-Order Execution

Code fragment

\[
\begin{align*}
\text{div} \quad & 10, 12, 14 \\
\text{addf} \quad & 110, 10, 18 \\
\text{multf} \quad & 17, 18, 14
\end{align*}
\]

Problem

- addf stalls due to RAW hazard
- multf stalls because addf stalls

In-order execution limits performance!

Instruction Re-ordering (Scheduling)

- Eliminate false control dependences
  - change order of instruction execution (statically or dynamically)
- Eliminate false data (storage) dependences
  - storage renaming
- Two problems/solutions go hand in hand

Dynamic vs. Static Scheduling

Solutions

- dynamic vs. static scheduling

Static scheduling (software)

- compiler reorganizes instructions
- simpler hardware
- can use more power algorithms
- Itanium, Crusoe, lots of DSP chips
Static Scheduling

Reorder code fragment

\[
\begin{align*}
\text{divf} & : f0, f2, f4 \\
\text{addf} & : f10, f0, f8 \\
\text{multf} & : f17, f8, f14 \\
\end{align*}
\]

Solution

- mult independent of addf and divf, so reorder
- addf still stalls

\[
\begin{align*}
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad 9 \\
\text{divf} & & & & & & & & \\
\text{addf} & & & & & & & & \\
\text{multf} & & & & & & & & \\
\end{align*}
\]

Dynamic Scheduling

Dynamic scheduling (hardware)

- handles dependences unknown at compile time
- reorganizes instructions!
- more complex hardware but code more portable
- “the real stuff”

- Shorter (more parallel) execution schedule while preserving illusion of sequential execution
- (Dynamic) instruction reordering algorithms
  - Scoreboarding
  - Tomasulo’s algorithm

Issue to Remember

- Data dependences: RAW (in registers, memory, condition codes, etc.)
- Name dependences: WAR and WAW
- Traps, interrupts, and exceptions
- Control dependences: branches and jumps

Dynamic Scheduling: Scoreboard

Centralized control scheme

- controls all instruction issue
- detects all hazards

Implemented in CDC6000 in 1964

- CDC 6000 has 18 separate functional units (not pipelined)
  - 4 FP: 2 multiply, 1 add, 1 divide
  - 7 memory units: 5 loads, 2 stores
  - 7 integer units, add, shift, logical, etc

Dynamic DLX:

- 2 FP mul, 1 FP add, 1 FP divide, 1 integer
Dynamic Pipeline with Scoreboard

Ignore IF and MEM for simplicity

4-stage execution

- Issue - check struct/WAW hazards, stall all issue till clear
- Read ops - check RAW, wait till operands ready, read regs
- Execute - execute operation, notify scoreboard when done
- Write - check for WAR, stall write until clear

Completing instr can’t write dest if an earlier instr hasn’t read dest

Scoreboard

3 data structures

- Instruction status
- Functional unit status
- Register result status

Instruction status

- Which execution stage the instruction is in

Scoreboard

Functional unit status

- Busy -- indicate whether functional unit is busy
- Op -- operation
- Fj -- destination register
- Fj, Fk -- source registers containing src1 and src2
- Qj, Qk -- functional units producing src1 and src2
- Rj, Rk -- ready bits for Fj, Fk

Register result status

- Indicates which functional unit (if any) will write the register
### Scoreboard Example

**Scoreboard rules**

<table>
<thead>
<tr>
<th>Status</th>
<th>Wait Until</th>
<th>Bookkeeping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td>not Busy[FU] &amp; not Result(D)</td>
<td>Busy[FU]=Yes; Op[DU]=op; F(FU)=D; F(FU)=S1; Rk[FU]=S2; Q(FU)=Result[S1]; OK(FU)=Result[S2]; R(FU)=not Q(FU); Rk(FU)=not Ok(FU); Result(D)=F(U);</td>
</tr>
<tr>
<td>Read Ops</td>
<td>Rj and Rk</td>
<td>Rj=No; Rk=No; Qj=0; Ok=0;</td>
</tr>
<tr>
<td>Execution done</td>
<td>FU complete</td>
<td></td>
</tr>
<tr>
<td>Write Result</td>
<td>if(F[R]) =&gt; F[FU] or Rj=No &amp; if(F[Rk]) =&gt; F[Du] or Rk=No))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if(Q[R])=FU then Rj=Yes;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if(Ok[R])=FU then Rk=Yes;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Result(F[FU])=0; Busy[FU]=No;</td>
<td></td>
</tr>
</tbody>
</table>

D= destination reg; S1 and S2 are source regs;

### Results

**Speedup**

- 1.7 for FORTRAN programs
- 2.5 for hand-coded assembly (what about modern compilers?)

**Hardware**

- scoreboard equal to one FU (main cost was busses)

**Limitations**

- no bypassing
- data dependences handled through registers
- WAW cause stalls (limits dynamic unrolling of loops)
Dynamic Scheduling

\[ DO \ i = 1, N \]
\[ C[i] = A[i] + s*B[i] \]

Assembly
\[
\begin{align*}
&\text{if } f0, A(r1) \\
&\text{if } f2, B(r1) \\
&\text{mul} f4, f2, f2 \\
&\text{add} f0, f2, f2 \\
&\text{str} f2, [C+r1]
\end{align*}
\]

What would scoreboard do?
Can we do better?

Dynamic Scheduling

Values need names
Registers provide names
But, registers also provide storage
Decouple name from storage
Think of registers as names or tags!

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)
- 360/85 introduced first cache in 1969

Pipelined rather than multiple functional units
- adder supports 3 instructions
- multiplier supports 2 instructions

Dynamic Pipeline with Tomasulo

Differences from scoreboard
- distributed hazard detection and control
- results are bypassed to functional units
- common data bus (CDB) for results

Tomasulo’s algorithm
- uses tags to identify data values
- reservation stations distribute control
- CDB broadcasts all results

Extend basic pipeline as example
- assume multiple FUs rather than pipelined
- main difference is register-memory instructions
**Tomasulo’s Algorithm**

3 major steps

1. **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

2. **Issue**
   - if not all operands are available, begin execution
   - if not, monitor CDB for operand

3. **Complete**
   - if CDB available, write result on CDB
   - if not, stall

**Tomasulo vs. Scoreboard**

Similar to scoreboard but
- no checking for WAW or WAR
- CDB broadcasts results to functional units, not just registers
Implementation

Reservation stations
- op – opcode
- Q, Q – tag fields (sources)
- V_j, V_k – operand values (sources)
- A – immediate value/effective address
- busy – currently in use

Register file and store buffer
- Q_i – tag field
- busy – currently in use

Tomasulo Code Example

if f6, 34(r2)
if f2, 45(r3)
mult f0, f2, f4
sub f8, f6, f2
div f10, f0, f6
add f6, f8, f2

Instruction Status (Illustration Only)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Execute</th>
<th>Write</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>

Tomasulo Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>V_j</th>
<th>V_i</th>
<th>Q_i</th>
<th>Q_j</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add1</td>
<td>Yes</td>
<td>SUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add2</td>
<td>Yes</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add3</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult</td>
<td>Yes</td>
<td>MULT</td>
<td>Reg[14]</td>
<td></td>
<td>Load2</td>
<td>Add1</td>
<td>Load2</td>
</tr>
</tbody>
</table>

Load buffers
Store buffers

<table>
<thead>
<tr>
<th>Field</th>
<th>Load</th>
<th>Load</th>
<th>Load</th>
<th>Field</th>
<th>Store</th>
<th>Store</th>
<th>Store</th>
</tr>
</thead>
</table>
Redo Tomasulo

```plaintext
loop:
  if f0, 0(r1)
  mult f4, f0, f2
  sfb 0(r1), f4
  sub r1, r1, 8
  bnez r1, loop
```

Tomasulo Details

Out-of-order loads and stores?
- what about WAW, RAW, WAR?
- compare all load addresses against
  - the addresses in store buffers
  - stall if they match

CDB is bottleneck
- duplicate
- increases required hardware

Complex implementation

Out-of-order loads and stores?
- what about WAW, RAW, WAR?
- compare all load addresses against
  - the addresses in store buffers
  - stall if they match

CDB is bottleneck
- duplicate
- increases required hardware

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 (important with only 4 regs!)
- requires many associative compares

Scoreboard vs. Tomasulo

<table>
<thead>
<tr>
<th>Feature</th>
<th>Scoreboard</th>
<th>Tomasulo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural</td>
<td>stall in Issue for FU</td>
<td>stall in Dispatch for RS, stall in RS for FU via CDB</td>
</tr>
<tr>
<td>RAW</td>
<td>via Registers</td>
<td></td>
</tr>
<tr>
<td>WAR</td>
<td>stall in Write</td>
<td>copy operand to RS</td>
</tr>
<tr>
<td>WAW</td>
<td>stall in Issue</td>
<td>Register renaming distributed</td>
</tr>
<tr>
<td>Logic</td>
<td>centralized</td>
<td></td>
</tr>
<tr>
<td>Bottleneck</td>
<td>no register bypass</td>
<td>one result bus</td>
</tr>
</tbody>
</table>

- stall in Issue, block)
Problem Created by OOO Execution

- Consider code fragment:
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>LF</td>
<td>F6, 34(R2)</td>
</tr>
<tr>
<td>LF</td>
<td>F2, 45(R3)</td>
</tr>
<tr>
<td>MULTF</td>
<td>F0, F2, F4</td>
</tr>
<tr>
<td>SUBF</td>
<td>F8, F6, F2</td>
</tr>
<tr>
<td>DIVF</td>
<td>F10, F0, F6</td>
</tr>
<tr>
<td>ADDF</td>
<td>F6, F8, F2</td>
</tr>
</tbody>
</table>

- What happens if DIVF causes an exception?
  ✓ ADDF has already completed

- What happens if the first LF has a page fault?
  ✓ MULTF and ADDF may have completed

- Out-of-order completion makes interrupts hard

Issue/Execution/Completion

- Issue
  ✓ send instruction to execution unit or reservation station

- Execution
  ✓ begin execution

- Completion
  ✓ change architectural state?
  ✓ make results visible

- In-order vs. Out-of-Order
  ✓ in-order means sequential order
  ✓ out-of-order means not sequential

Solutions

- Imprecise interrupts
  ✓ ignore the problem
  ✓ makes page faults difficult
  ✓ IEEE FP standard strongly suggests precise interrupts

- In-order completion
  ✓ stall pipeline when necessary

- Software cleanup
  ✓ save information for trap handlers
  ✓ machine dependent

Solutions, contd.

- Hardware cleanup
  ✓ restore to consistent state

- Re-order instructions
  ✓ complete out of order
  ✓ commit or retire in order
  ✓ issue, execute, complete, commit (or retire)
**Smith and Pleszkun paper**

- Several schemes to maintain precise interrupts
- Architecture with multiple functional units
- Scheduling result bus (RSR)
- Re-order buffer
- Re-order buffer with bypass
- History buffer
- Future file

**Scheduling Result Bus**

- Result Shift Register (RSR) to control result bus
  - at most one completion per cycle
  - Instruction with latency reserves stage i when it issues
    - stalls if stage is taken
  - Shift down each cycle
- Instruction in stage 1 gets result bus

<table>
<thead>
<tr>
<th>Stage</th>
<th>FU</th>
<th>Data</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>FADD</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>FADD</td>
<td>4</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**In-order Completion**

- Reserve preceding stages
  - instruction taking j stages reserves all available stages i<j
  - prevents later (sequential) instruction from completing before
- Must synchronize stores (stores do not use result bus)
  - wait for RSR to empty
  - put dummy store in RSR
- What if store causes exception??

**Re-order Buffer (ROB)**

- Overview
  - instructions complete out of order
  - re-order buffer reorganizes instructions
  - modify state in order
- Add tag to RSR to point to corresponding RB entry
Re-Order Buffer

- On dispatch, allocate ROB entry and RSR entry
  - make RSR entry point to ROB entry
- On completion, put result in ROB entry pointed to by RSR tag
  - make ROB entry valid
- When head of ROB is valid, commit
  - update register and exceptions

Re-Order Buffer with Bypass

- ROB delays update of register file
  - aggravates dependences
- Bypass paths needed to alleviate aggravation of dependences

History Buffer

- Overview
  - instructions complete out of order
  - register file updated immediately
  - hardware cleans up on exception
- History Buffer (HB)
  - logs previous value of register
    - used to undo instructions

History Buffer

- On issue, copy current value of dst reg to HB
- On completion, mark HB entry valid
- On exception, mark exception bit
- When head of HB contains valid entry, reuse entry
- When entry has exception
  - stall issue
  - wait for pipeline to empty
  - roll back register file from tail to head
  - PC at head is precise PC
- Hardware
  - 3 read ports
Future File (FF)

- Overview
  - separate architectural and future register files
  - future file is update out of order
  - architectural file is updated in order
  - ROB controls updates to architectural file

Future File

- Future file is managed like an ordinary imprecise pipeline
- Architectural file managed like ROB scheme
  - update file when head is valid
- On exception
  - architectural file contains precise state
- Advantages
  - state saving is easy
  - no extra bypassing

Checkpoint Repair

- Make checkpoint of machine state at certain event
  - e.g., when branch is encountered
- On exception, roll back to checkpoint and restart
- Very similar to history file

Summary of Mechanisms

- History Buffer/Future File/Checkpoint Repair: no unnecessary aggravation of dependences, but need to "back up"
- Reorder buffer: dependences can be aggravated unless bypass provided
  - OOO techniques can help tolerate aggravated dependences
- Later, ROB can (trivially) be extended to support speculation
**What Has Been Done So Far**

- Given a window of instructions that need to be executed (created by sequencing through a static program) how to build a schedule for efficient execution?
- Nature/shape of schedule depends upon
  - operation latencies
  - number of instructions available
  - (artificial or real) dependence relationships
- Techniques to reorder instructions (overcome “artificial” dependences) and allow more parallel schedule
- Techniques to give appearance of sequential execution
- Next step is to develop techniques to further decrease constraints/increase parallelism

**Further Facilitation of Parallel Schedule**

- Increase the size of the instruction window; try to overcome true control dependences
- Increase scheduling flexibility by decreasing the number of dependences: try to overcome “true” data dependences

**Overcoming “True” Control Dependences**

- *Conditional or Guarded* execution: convert control dependences to data dependences
  - help static code arrangement
- *Speculative* execution: guess “value” of control condition
  - condition can take on only 2 values: True or False

**Speculative Execution**

- Try to overcome “true” dependences
- Guess outcome of dependence
- If guess is correct, great!!
- If guess is incorrect, recover from incorrect guess
  - need to separate speculative and non-speculative values
- Can be done in software or hardware
  - software speculation later
  - hardware speculation now
**Hardware/Dynamic Speculation**

- Two main types
  - control speculation
  - data speculation
- Many variants of basic types in use or being studied currently

**Control Speculation**

- Execute instruction before its control condition
- Guess value of control condition
  - control condition is Boolean variable
  - a.k.a branch prediction
- Allows more instructions to be present in instruction window
  - allows better schedules to be developed.

**Data Speculation**

- Execute instruction before its data dependencies are resolved
- Guess value of (non-Boolean) variable
- Many forms possible (load/store address speculation, data value speculation, etc.)
- Next step in overall progression
- More later

**Requirements for Speculative Execution**

- Techniques to guess/predict value of variable/condition.
- Distinguish between non-speculative and speculative state
- Ability to “back out” from speculation, i.e., recover non-speculative state until point of speculation.
Initiating Control Speculation

- Guess which direction the branch will take
  - Branch prediction
- Guess where branch will go
  - Branch target prediction
- Techniques for the above can be separate or combined

Dynamic Branch Prediction

- Basic idea
  - have hardware try to guess whether and where a branch instruction will go
- Example
  - address instrn
    ------- -------
    0x64    BNEZ R1, +4
    ...    ...
    0x74    ADD R3, R2, R1
- Start with Branch PC (0x64) and produce
  - Prediction (T=taken, N=not taken)
  - Prediction + Target PC (0x74)
  - Prediction + Target PC + Target Instrn. (ADD R3, R2, R1)

Branch History Table

- Start with Branch PC and produce Prediction (T=taken, N=not taken)
- Example implementation: 2**m x 1 memory
  - read prediction with m LSBs of branch PC
  - change bit on misprediction
  - may use bit from wrong PC (aliasing)
- Better predictors
  - 2-bit state machine
  - correlating predictors

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
Two-Bit Counters

Branch History Table

Branch History Table

Branch History Table

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 (approximation to path followed):

- Use branch history register (BHR) to hold recent outcomes
- Use both PC and BHR to access table
- Recent proposals keep more complete path information
Two-Level Predictors (Yeh & Patt)

PC and BHR can be combined as:

- Concatenated
- Completely overlapped
- Partially overlapped

PAp predictor
PAg predictor
GAg predictor

PAp

Per-address history:
- Branches correlate with their own history
- Also, referred to as "local"

Per-address patterns:
- No conflicts among branches

What branch behavior do they predict?

PAg

Per-address history:
- Another local predictor

Global patterns:
- Take advantage of similar loop patterns
- Save space

GAg

Global history:
- Branch history correlates among all branches
- Also, referred to as "global"

Global patterns:
- Conflicts from all over the program
**Gshare (McFarling)**

- Reduces the conflicts among program phases
- Distributes patterns across tables
- Uses hashing of PC

**Branch Target Buffer**

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?

**Branch Target Cache**

Start with branch PC and produce

- prediction + target PC + target instruction

E.g. implementation: a special cache

- index and tags: branch PC
- data: prediction info + target PC + one or more target instr

Could send target instruction rather than branch into ID

- zero cycle branch: “branch folding”
Branch Target Cache

Considerations

- Many more bits per entry than branch target buffer
- Size and associativity

Call Return Stack

Subroutine return

- Use call return stack (in hardware)

General indirect jumps

- More than one target -> not just T/NT
- Path-based schemes may have potential (later?)

More Advanced Techniques

20 years worth of research
Recent technique: Trace Cache
- Keep a path of instruction traces
- Fold (remove) branches
Also, in case studies:
- Hybrid predictors (local and global) in 21264

Dynamic Branch Prediction Summary

- Considerations
  - Results always correct (e.g., squash mispredicted instms)
  - Don't slow cycle time (much)
  - Very fast on correct predictions
  - High prediction accuracy
  - Not too slow on incorrect predictions
- Bottom line
  - Useful for single-issue pipelines, i.e., pipelined schedules
  - Critical for multiple-issue machines, i.e., parallel-pipelined, OOO schedules
Where do we stand?

- Have predicted control path
- Instructions from predicted path included in "scheduling window"
- Schedule will be composed of non-speculative and speculative instructions
  - Have not figured out how to schedule instructions for execution yet, i.e., how to implement schedule
- Question: How to distinguish between speculative and non-speculative values?

Dealing with Speculative State

- Need more physical storage/registers than logical registers
  - physical storage used to hold results of (speculative and non-speculative) instructions
- Need mechanisms to map logical (architectural) registers to physical storage elements
- Need mechanisms distinguish between speculative and non-speculative state

Storage in a Superscalar Processor

- The places data may potentially reside:
  - registers
  - reservation stations
  - within functional units
  - reorder buffer

- Registers:
  - May be logical = physical, or
  - logical != physical; i.e. renaming

- Reservation Stations:
  - may be FIFO or random access
  - may be unified or partitioned by units

Reservation Stations

- Need more physical storage/registers than logical registers
- physical storage used to hold results of (speculative and non-speculative) instructions
- Need mechanisms to map logical (architectural) registers to physical storage elements
- Need mechanisms distinguish between speculative and non-speculative state
Register Renaming

- More physical registers than logical registers
  - Physical "registers" can be implemented in a variety of ways
    (e.g., reservation stations, "traditional registers", reorder buffer)
- Renaming associates a physical register with a logical
  register (at a particular point in time)
  - same logical register can be mapped onto different physical
    registers, including multiple physical registers
- Recall Tomasulo algorithm
  - renaming changed logical register name to "name", i.e., tag of
    reservation station
- Re-naming avoids WAW and WAR hazards; turns instruction
  stream into "single assignment" form

Tomasulo’s Algorithm

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

POWER1 (RS6000)’s Method

At dispatch, rename registers
Instruction issue queues
  - dynamic issue across (not within) queues
Conventional dataflow:
  - only between registers & FUs
Precise state maintained in piecemeal fashion
  - condition registers use history file
  - integer intrs complete in order (FP not allowed to pass int)
  - floating point exceptions are imprecise

POWER 1’s Method Illustrated

- More physical registers than logical registers
- Renaming avoids WAW and WAR hazards
- Turns instruction stream into "single assignment" form
- Assume 8 logical, 16 physical registers
Register Renaming Example

- Example
  LF   F6, 34 (R2)
  LF   F2, 45 (R3)
  MULTF  F0, F2, F4
  SUBF  F7, F6, F2
  DIVF  F1, F0, F6
  ADDF  F6, F7, F2

- Assume 8 logical, 16 physical registers

Register Renaming Example, contd.

Register mapping before:
Logical   Physical
0        12      0   2
1        13      1   4
2        15      2   1
3        14      3   14
4        9       4   9
5        7       5   7
6        6       6   5
7        8       7   3

Free Pool: 0, 1, 2, 3, 4, 5, 10, 11

Register mapping after:
Logical   Physical
0        0       0   2
1        13      1   4
2        2       2   1
3        3       3   14
4        4       4   9
5        5       5   7
6        6       6   5
7        7       7   3

Free Pool: 10, 11, ?, ?, ?

Renaming Example, contd.

Code with physical register assignments

LF   P0, 34 (R2)
LF   P1, 45 (R3)
MULTF  P2, P1, P0
SUBF  P3, P0, P1
DIVF  P4, P2, P0
ADDF  P5, P3, P1

Register Renaming, contd.

- Reclaiming registers (returning to free pool)
  - Ignoring precise interrupts, a register can be returned after the last read is done.
  - Considering precise interrupts, a register can be returned after it is logically over-written
  - Can be done in a number of ways, e.g. via a counter per register.
  - Later, we will see a straightforward way.
**Sohi's Method**

Logical registers = physical registers
Reorder buffer: (1) commits state in order, (2) holds renamed registers
Reservation stations are randomly accessed
Registers read only at dispatch
Values come from: (1) reorder buffer, (2) functional units

**Sohi Method Example**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Entry Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>LF F6,34(R2)</td>
<td>Reorder buffer while FP divide in progress:</td>
</tr>
<tr>
<td>LF F2,45(R3)</td>
<td>Entry Content</td>
</tr>
<tr>
<td>MULT F0,F2,F4</td>
<td>0 F6</td>
</tr>
<tr>
<td>SUBF F7,F6,F2</td>
<td>1 F2</td>
</tr>
<tr>
<td>DIVF F1,F0,F6</td>
<td>2 F0</td>
</tr>
<tr>
<td>ADDF F6,F7,F2</td>
<td>3 F7</td>
</tr>
<tr>
<td></td>
<td>4 F1</td>
</tr>
<tr>
<td></td>
<td>5 F6</td>
</tr>
</tbody>
</table>

**MIPS R10000’s Method**

Uses logical registers != physical registers (like POWER1)
Data movement only between regs and units
Reorder buffer used for control
  - register reservations
  - returning physical registers to free pool
Register reservation bits
  - apply to physical registers
  - monitored by instructions in RS (replaces tag compares)

**MIPS R10000 Illustrated**
MIPS R10000: Reclaiming Registers

Reclaiming physical registers

- reorder buffer contents:
  - PC, errors, result register, previous mapping
  - previous physical register mapping is copied to the ROB
  - when a new mapping is made
  - when an instruction is committed from the reorder buffer
  - the old physical register is no longer needed
  - so can be reclaimed

Speculative Execution

Predict branches and speculatively execute instructions

1. Correct: great!
2. Incorrect:
   - Squash later speculative instructions: How?
   - Exactly like precise interrupts

Reorder buffer can be used for squashing

- add "speculative bit" to reorder buffer
- do not commit any instruction if speculative bit is set
- correct \(\Rightarrow\) clear speculative bit
- incorrect \(\Rightarrow\) throw away speculative instrs in reorder buffer
- remove using "old mapping information" to restore map

Example of Speculative Execution

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ld</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Ld</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Add</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Add</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BNEZ</td>
<td>13</td>
<td>14</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ld</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Ld</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Add</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Speculative Example, contd.

Register mapping at time of speculative branch:

Free pool at start: 0,2,3,4,5,6,8,15

Logical Physical

- 0 3
- 1 7
- 2 4
- 3 13
- 4 14
- 5 2
- 6 0
- 7 12

Reorder buffer:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ld</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Ld</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Add</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Add</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BNEZ</td>
<td>13</td>
<td>14</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ld</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Ld</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Add</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Summary of Speculative Execution

- Initiate speculation with branch prediction
- Allow scheduler to schedule speculative and non-speculative instructions at same time
- Distinguish between speculative and non-speculative state
- Use a "precise state" mechanism to recover from misspeculation
- History file variant used in Motorola 88110
- Reorder buffer variants used in most other processors (PPC 604, 620, Intel Pentium Pro, AMD K5, MIPS R1000, HP PA8000, and many others)

### Scheduling Instructions

- How to send them from their "holding stations" to the execution units for processing?
- Need to know which instructions are ready
- Need to select from ready instructions
- Need to "wake up" instructions as instructions complete

### Holding Stations/Queue Types

- **Wakeup**
  - Update availability status of operands
- **Ready**
  - Monitor availability status of operands
- **Select instruction from queue**
  - select instruction at head of queue
  - arbitrary selection within queue

---

### Instruction Scheduling
Example

r1 ← loop length
r2 ← 0
r4 ← addb(c)
r5 ← addr(c)
r6 ← addr(c)

loop: r3 ← mem(r4+r2)
r7 ← r7 + r3
r1 ← r1 - 1
mem(r6+r2) ← r7
r2 ← r2 + 8
BNEZ r1, loop

Example: Dynamic Code

r3 ← mem(r4+r2)
r7 ← mem(r5+r2)
r7 ← r7 + r3
r1 ← r1 - 1
mem(r6+r2) ← r7
r2 ← r2 + 8
BNEZ r1, loop

Example: After Branch Prediction

r3 ← mem(r4+r2)
r7 ← mem(r5+r2)
r7 ← r7 + r3
r1 ← r1 - 1
mem(r6+r2) ← r7
r2 ← r2 + 8
r3 ← mem(r4+r2)
r7 ← mem(r5+r2)
r7 ← r7 + r3
r1 ← r1 - 1
mem(r6+r2) ← r7
r2 ← r2 + 8

Example: After Register Renaming

p6 ← mem(p1+p4)
p4 ← mem(p1+p4)
p7 ← p4 + p6
p8 ← p6 - 1
mem(p1+p4) ← p7
p9 ← p4 + 0
p10 ← mem(p1+p3)
p11 ← mem(p1+p3)
p12 ← p10 + p11
p13 ← p8 - 1
mem(p1+p3) ← p12
p14 ← p8 + 0
p15 ← mem(p1+p4)
p16 ← mem(p1+p4)
p17 ← p16 + p14
p18 ← p14 - 1
mem(p1+p4) ← p17
p19 ← p14 + 0
Dealing with Memory Operations

- Why are memory operations different?
- Progression of "schedule improvement" for memory operations
  □ Recall progression for register operations

Reg. And Memory Operation Differences

- Storage (register) accessed by register operation is known when instruction is decoded.
- Storage (memory location) accessed by memory operation is known only after address calculation is performed.
- Split memory operation into two parts: (i) address calculation, and (ii) performing actual operation.

Memory Operation Schedule Improvement

- Overlap memory operations (non-blocking caches)
- Multiple memory accesses.
- OOO memory operations
- Above scenarios are similar to what we had for registers, but relative order may be different
- Relative frequencies impact need for particular technique

OOO Memory Operations

- Load/Store operations submitted to memory via queues
  □ Load Address Queue
  □ Store Address Queue
  □ Store Data Queue
- Load and store addresses computed in order (for hazard detection)
  □ => load/store reservation stations are FIFO
  □ store issues waits only for address register
  □ store address and data are matched later in a store buffer
  □ store is committed via reorder buffer mechanism (or other precise interrupt mechanism)
OOO Memory Operations

- Loads can pass earlier loads (if Load Address Queue is non-FIFO)
- Loads can pass earlier stores, if store address known
- Unknown store address stalls all later memory operations
- Stores can’t pass earlier stores
  - no WAW hazard resolution

Future OOO Memory Operations

- Load and store addresses not computed in order (data dependence speculation)
  - Stores can pass earlier loads
  - Stores can pass earlier stores
- Recover in case of mis-speculation (just like recovery for control mis-speculation)
- Multiple stores to same location (memory renaming)
- Passing values (speculatively) from stores to loads (memory bypassing)

Put it All Together

- Multiple issue, typically coupled with out-of-order execution of instructions. Must include the following:
  - High performance instruction fetch strategies
    - Multiple instructions per cycle
    - Branch prediction
  - Methods for resolving dependences
    - Eliminate "false" dependences (WAR, WAW)
    - Set up linkages for true dependences, e.g. register renaming

Put it All Together

- Methods for parallel issue
  - Parallel resources:
    - functional units, paths, register ports
    - High performance memory systems
- Methods for committing processor state correctly (Precise interrupts)
**Elements of Advanced Superscalar**

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**

[Diagram of a superscalar processor]

**The Big Picture**

<table>
<thead>
<tr>
<th>Program Form</th>
<th>Processing Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static program</td>
<td>Instruction fetch &amp; branch prediction</td>
</tr>
<tr>
<td>Dynamic instruction stream</td>
<td>Dependence checking &amp; dispatch</td>
</tr>
<tr>
<td>Execution window</td>
<td>Instruction issue</td>
</tr>
<tr>
<td>Completed</td>
<td>Instruction execution</td>
</tr>
</tbody>
</table>

**Elements of Advanced Superscalar**

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
**Instruction Fetching**

Multiple instructions must be fetched per cycle
IBM POWER solution e.g., RS6000 (simplified)
A branch can initiate fetching anywhere in a line
 contiguous lines are in an odd/even pair

---

**More on Instruction Fetching**

- With wider issue, multiple instruction fetching may span multiple contiguous cache lines
- With wider issue, multiple instruction fetching may span multiple non-contiguous cache lines
- May include multiple branch instructions in fetched block ➔ multiple branch predictions needed
- More complex (wider) instruction fetching schemes still subject of research ➔ trace caches

---

**RS6000 I-fetch**

---

**Dealing with Branches**

- Recognizing branches
  ➔ Can't wait until decode (lost slots)
  ➔ Use "predecode" info with cache line
  ➔ Predecode used to initiate other actions associated with branch (e.g., prediction)
- Determining branch outcome
  ➔ Predict outcome using predictor
- Computing branch targets, i.e., where does branch go?
  ➔ Use branch target cache
- Transferring control
  ➔ use stockpiles to smoothen flow
Where do we stand and what is next?

- Multiple instructions fetched
- Decode and set up dependence links
  - Identify storage element where result of operation is going to reside (was logical register)
  - Identify storage elements that contain values for source operands (were logical registers)
- Dispatch instructions to logic (issue logic) that will develop the execution schedule
  - Need storage to "hold" instructions while they are waiting to be issued

Decode and Dependence Linkage

- Parallel decoders used to decode multiple instructions
  - Decoder has recurrence (prefix problem)
  - Typically O(n^2) crosschecks
- Register renaming method used to set up dependence links
- Instructions forwarded to appropriate holding stations to await execution

Case Study: MIPS R10000

- 4-way superscalar - 1 memory, 2 ALU, 2 FP
- Register renaming (32 logical -> 64 physical)
- Reorder buffer
- 4-deep branch speculation
- Resume cache:
  - Keeps instructions on non-predicted path for fast recovery
- 2-way associative, 32 KB instruction and data caches
Case Study: Alpha 21164

4-way superscalar - 2 memory, 2 ALU, 2 FP
issue in-order
1-deep branch speculation
direct-mapped 8 KB instruction and data caches
on-chip L2-cache

Case Study: Intel P6

CISC to ROP conversion
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, high branch penalty
First implementation of Yeh&Patt predictor

CISC to ROP Conversion

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 already critical

Superscalar Trends
Wire delays don’t change with smaller feature sizes
1. smaller faster transistors
2. larger dies
=> longer wires on-chip will take multiple clocks
So on-chip communication, not computation will dominate
Case Study: Alpha 21264

6-way superscalar – 4 ALU + memory, 2 FP
issue out-of-order, retire in-order (precise)
• 2-way cluster, 1-cycle bypass
7-deep branch speculation
• Hybrid 1K local + 4K global (PAg & GAg)
2-way 64 KB instruction and data caches
• Wave-pipelined
• Line & way prediction to reduce latency
• off-chip L2-cache