EE382A Lecture 6:
Register Renaming

Department of Electrical Engineering
Stanford University
http://eeclass.stanford.edu/ee382a
Announcements

• Project proposal due on Wed 10/14
  – 2-3 pages submitted through email
  – List the group members
  – Describe the topic including why it is important and your thesis
  – Describe the methodology you will use (experiments, tools, machines)
  – Statement of expected results
  – Few key references to related work

• Still missing some photos
Lecture 6 Outline

1. Branch Prediction (epilog)
   a. 2-level Predictors
   b. AMD Opteron Example
   c. Confidence Prediction
   d. Trace Cache

2. Register Data Flow
   a. False Register Dependences
   b. Register Renaming Technique
   c. Register Renaming Implementation
Dynamic Branch Prediction Using History

- **nPC to Icache**

- **nPC(seq.) = PC+4**

- **nPC = BP(PC)**

- **Branch Predictor (using a BTB)**

- **FA-mux**

- **Decode Buffer**

- **BTB update (target addr. and history)**

- **Dispatch Buffer**

- **Dispatch**

- **Issue**

- **Reservation Stations**

- **Completion Buffer**

**Flowchart Details:**
- The diagram illustrates the process flow of dynamic branch prediction using history, including fetch, decode, dispatch, execution, and completion stages.
- Branch prediction is indicated at multiple points in the flowchart.
- The speculative target and condition are marked in the diagram.
- The BTB (Branch Target Buffer) update is shown with target address and history information.
- The execution flow transitions through reservation stations and completion buffer.
2-Level Adaptive Prediction [Yeh & Patt]
Nomenclature: \{G,P\}A\{g,p,s\}

To achieve 97% average prediction accuracy:

- \( G (1) \) BHR: 18 bits; \( g (1) \) PHT: \( 2^{18} \times 2 \) bits \( total = 524 \) kbits
- \( P (512 \times 4) \) BHR: 12 bits; \( g (1) \) PHT: \( 2^{12} \times 2 \) bits \( total = 33 \) kbits
- \( P (512 \times 4) \) BHR: 6 bits; \( s (512) \) PHT: \( 2^6 \times 2 \) bits \( total = 78 \) kbits
Example: Global BHSR Scheme (GAs)

Branch Address

j bits

Branch History Shift Register (BHSR)

k bits

BHT of $2 \times 2^{i+k}$
Example: Per-Branch BHSR Scheme (PAs)

Branch Address

Standard BHT

Branch History Shift Register (BHSR) $k \times 2^i$

Prediction

BHT of $2 \times 2^{i+k}$
Gshare Branch Prediction [McFarling]

Branch Address

\[ j \text{ bits} \]

\[ \text{xor} \]

Branch History Shift Register (BHSR)

\[ k \text{ bits} \]

\[ \text{BHT of } 2 \times 2^{\max(j,k)} \]

Prediction
Fetch & Predict Example: AMD Opteron
Why is Prediction Important in Opteron?

![Diagram showing the comparison between Hammer and Athlon in terms of instruction decode and execution stages, highlighting the mispredict penalty for each.]
Fetch & Predict Example: AMD Opteron
Other Branch Prediction Related Issues

- **Multi-cycle BTB**
  - Keep fetching sequentially, repair later (bubbles for taken branches)
  - Need pipelined access though

- **BTB & predictor in series**
  - Get fast target/direction prediction from BTB only
  - After decoding, use predictor to verify BTB
    - Causes a pipeline mini-flush if BTB was wrong
  - This approach allows for a much larger/slower predictor

- **BTB and predictor integration**
  - Can merge BTB with the local part of a predictor
  - Can merge both with I-cache entries

- **Predictor/BTB/RAS updates**
  - Can you see any issue?
Prediction Confidence
A Very Useful Tool for Speculation

• Estimate if your prediction is likely to be correct

• Applications
  – Avoid fetching down unlikely path
    • Save time & power by waiting
  – Start executing down both paths (selective eager execution)
  – Switch to another thread (for multithreaded processors)

• Implementation
  – Naïve: don’t use NT or TN states in 2-bit counters
  – Better: array of CIR (correct/incorrect registers)
    • Shift in if last prediction was correct/incorrect
    • Count the number of 0s to determine confidence
  – Many other implementations are possible
    • Using counters etc
Branch Confidence Prediction

Diagram:
- Branch address
- Global BHR
- XOR
- Table of CIRs
- Reduction function
- Confidence prediction
Dynamic History Length

- Four types of history
  - Local (bimodal) history (Smith predictor)
    - Table of counters summarizes local history
    - Simple, but only effective for biased branches
  - Local outcome history
    - Shift register of individual branch outcomes
    - Separate counter for each outcome history
  - Global outcome history
    - Shift register of recent branch outcomes
    - Separate counter for each outcome history
  - Path history
    - Shift register of recent (partial) block addresses
    - Can differentiate similar global outcome histories
- Can combine or “alloy” histories in many ways
Understanding Advanced Predictors

- History length
  - Short history—lower training cost
  - Long history—captures macro-level behavior
  - Variable history length predictors

- Really long history (long loops)
  - Loop count predictors
  - Fourier transform into frequency domain

- Limited capacity & interference
  - Constructive vs. destructive
  - Bi-mode, gskewed, agree, YAGS
  - Read sec. 9.3.2 carefully
High-Bandwidth Fetch: Trace Cache

- Fold out taken branches by *tracing* instructions as they commit into a *fill buffer*
Intel Pentium 4 Trace Cache

- No first-level instruction cache: trace cache only
- Trace cache BTB identifies next trace
- Miss leads to fetch from level two cache
- Trace cache instructions are decoded (uops)
Modern Superscalar, Out-of-order Processor

- Pipelining reduces cycle time
- Superscalar increases IPC (instruction per cycle)
- Both schemes need to find lots of ILP in the program
  - Must simultaneously increase number of instructions considered, number of instructions executed, and allow for out-of-order execution
What Limits ILP

INSTRUCTION PROCESSING CONSTRAINTS

- Resource Contention (Structural Dependences)
- Control Dependences
- Code Dependences
- Data Dependences
- Storage Conflicts
- Output Dependences (WAW)
- (WAR) Anti-Dependences
- (RAW) True Dependences
Register Renaming & Dynamic Scheduling

• Register Renaming: address limitations of the scoreboard
  – Scoreboard limitation
    • Up to one pending instruction per destination register
  – Eliminate WAR and WAW dependences without stalling

• Dynamic scheduling
  – Track & resolve true-data dependences (RAW)
  – Scheduling hardware:
    • Instruction window, reservation stations, common data bus, …
  – Original proposal: Tomasulo’s algorithm [Tomasulo, 1967]
Register Data Flow

Each ALU Instruction:

\[ \text{Ri} \leftarrow F_n (\text{Rj}, \text{Rk}) \]

Dest.  Funct.  Source  Registers
Reg.    Unit     Registers

“Register Transfer”

INSTRUCTION EXECUTION MODEL

 Registers
\[ \begin{array}{c}
\text{R0} \\
\text{R1} \\
\ldots \\
\text{Rm}
\end{array} \]

FU\_1  FU\_2  \ldots  FU\_n

Functional Units

Interconnect

“Read”  “Execute”  “Write”

Need Availability of F\_n (Structural Dependences)
Need Availability of Rj, Rk (True Data Dependences)
Need Availability of Ri (Anti-and output Dependences)
Causes of (Register) Storage Conflict

REGISTER RECYCLING

MAXIMIZE USE OF REGISTERS
MULTIPLE ASSIGNMENTS OF VALUES TO REGISTERS

OUT OF ORDER ISSUING AND COMPLETION

LOSE IMPLIED PRECEDENCE OF SEQUENTIAL CODE
LOSE 1-1 CORRESPONDENCE BETWEEN VALUES AND REGISTERS
The Reason for WAW and WAR: Register Recycling

**COMPILER REGISTER ALLOCATION**

**CODE GENERATION**

**REG. ALLOCATION**

**INSTRUCTION LOOPS**

```plaintext
For (k=1; k<= 10; k++)
    t += a[i][k] * b[k][j];
```

Reuse Same Set of Reg. in each Iteration

Overlapped Execution of different Iterations
Resolving False Dependences

Must Prevent (2) from completing before (1) is dispatched

\[
(1) \quad R4 \leftarrow R3 + 1 \\
(2) \quad R3 \leftarrow R5 + 1
\]

Must Prevent (2) from completing before (1) completes

\[
(1) \quad R3 \leftarrow R3 + R5 \\
\quad \leftarrow R3 \\
\quad \leftarrow R3 \\
(2) \quad R3 \leftarrow R5 + 1
\]

Stalling: delay dispatching (or write back) of the later instruction

Copy Operands: Copy not-yet-used operand to prevent being overwritten (WAR)

Register Renaming: use a different register (WAW & WAR)
Register Renaming: The Idea

- Anti and output dependences are false dependences

\[ r_3 \leftarrow r_1 \text{ op } r_2 \]
\[ r_5 \leftarrow r_3 \text{ op } r_4 \]
\[ r_3 \leftarrow r_6 \text{ op } r_7 \]

- The dependence is on name/location rather than data

- Given unlimited number of registers, anti and output dependences can always be eliminated

Original

- \( r_1 \leftarrow r_2 / r_3 \)
- \( r_4 \leftarrow r_1 * r_5 \)
- \( r_1 \leftarrow r_3 + r_6 \)
- \( r_3 \leftarrow r_1 - r_4 \)

Renamed

- \( r_1 \leftarrow r_2 / r_3 \)
- \( r_4 \leftarrow r_1 * r_5 \)
- \( r_8 \leftarrow r_3 + r_6 \)
- \( r_9 \leftarrow r_8 - r_4 \)
Register Renaming Technique

Register Renaming Resolves:

- Anti-Dependences
- Output Dependences

<table>
<thead>
<tr>
<th>Architected Registers</th>
<th>Physical Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>P1</td>
</tr>
<tr>
<td>R2</td>
<td>P2</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Rn</td>
<td>Pn</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td>Pn + k</td>
</tr>
</tbody>
</table>

Design of Redundant Registers

- Number:
  - One
  - Multiple

- Allocation:
  - Fixed for Each Register
  - Pooled for all Registers

- Location:
  - Attached to Register File (Centralized)
  - Attached to functional units (Distributed)
Register Renaming Implementation

- Renaming:
  - Map a small set of architecture registers to a large set of physical registers
  - New mapping for an architectural register when it is assigned a new value

- Renaming buffer organization (how are registers stored)
  - Unified RF, split RF, renaming in the ROB
  - RF = register file

- Number of renaming registers
- Number of read/write ports
- Register mapping (how do I find the register I am looking for)
  - Allocation, de-allocation, and tracking
Renaming Buffer Options

- Unified/merged register file – MIPS R10K, Alpha 21264
  - Registers change role architecture to renamed
- Rename register file (RRF) – PA 8500, PPC 620
  - Holds new values until they are committed to ARF
  - Extra data transfer…
- Renaming in the ROB – Pentium III
- Note: can have a single scheme or separate for integer/FP
Unified Register File:
Physical Register FSM

Available

Renamed
Not Valid

Architectural

Renamed
Valid

Initialized
(remaining registers)

New allocation
on instruction decode

Instruction Cancelled

Instruction Completes Execution

Deallocate when an overwriting Instruction Retires

Instruction Retires

Initialized
(first 32 registers)
Number of Rename Registers

• Naïve: as many as the number of pending instructions
  – Waiting to be scheduled + executing + waiting to commit

• Simplification
  – Do not need renaming for stores, branches, …

• Usual approach:
  – # scheduler entries ≤ # RRF entries ≤ # ROB entries

• Examples:
  – PPC 620: scheduler 15, RRF 16 (RRF), ROB 16
  – MIPS R12000: scheduler 48, RRF 64 (merged), ROB 48
  – Pentium III: scheduler 20, RRF 40 (in ROB), ROB 40
Register File Ports

- Read: if operands read as instructions enter scheduler
  - Max # ports = 2 * # instructions dispatched
- Read: if operands read as instruction leave scheduler
  - Max # ports = 2* # instructions issued
    - Can be wider than the # of instructions dispatched…
- Write: # of FUs or # of instructions committing
  - Depends on unified vs separate rename registers

- Notes:
  - Can implement less ports and have structural hazards
    - Need control logic for port assignment & hazard handling
  - When using separate RRF and ARF, need ports for the final transfer
  - Alternatives to increasing ports: duplicated RF or banked RF
    - What are the issues?
Register Mapping
(From Architectural to Physical Address)

- Option 1: use a map table (ARF # → physical location)
  - Map holds the state of the register too...
  - Simple, but need two steps for reading (ok if operands read late)
- Option 2: associative search in RRF, ROB, ...
  - Each physical register remembers its status (ok if operand read early)
  - More complicated but one step read
Integrating Map Tables with the ARF

(a)

(b)
Renaming Operation: Allocation, Lookup, De-allocation

• At dispatch: for each instruction handled in parallel
  – Check the physical location & availability of source operands
  – Map destination register to new physical register
    • Stall if no register available
  – Note: must have enough ports to any map tables

• At complete: update physical location

• At commit/retire: for each instruction handled in parallel
  – Copy from RRF/ROB to ARF & deallocate RRF entry OR
  – Upgrade physical location and deallocate register with old value
    • It is now safe to do that

• Question: can we allocate later or deallocate earlier?
Renaming Operation
Renaming Difficulties: Wide Instruction Issue

• Need many ports in RFs and mapping tables

• Instruction dependencies during dispatching/issuing/committing
  – Must handle dependencies across instructions
  – E.g. add R1 ← R2 + R3; sub R6 ← R1 + R5

  – Implementation: use comparators, multiplexors, counters
    • Comparators: discover RAW dependencies
    • Multiplexors: generate right physical address (old or new allocation)
    • Counters: determine number of physical registers allocated
Renaming Difficulties: Mispredictions & Exceptions

• If exception/misprediction occurs, register mapping must be precise

• Separate RRF: consider all RRF entries free

• ROB renaming: consider all ROB entries free

• Unified RF: restore precise mapping
  – Single map: traverse ROB to undo mapping (history file approach)
    • ROB must remember old mapping…
  – Two maps: architectural and future register map
    • On exception, copy architectural map into future map…
  – Checkpointing: keep regular check points of map, restore when needed
    • When do we make a checkpoint? On every instruction? On every branch?
    • What are the trade-offs?
    • We’ll revisit this approach later on…
Dynamic Scheduling Based on Reservation Stations

Diagram:
- Dispatch Buffer
- Dispatch
- Reg. File
- Ren. Reg.
- Reg. Write Back
- Reservation Stations
- Allocate Reorder Buffer entries
- Branch
- Integer
- Integer
- Float.-Point
- Load/Store
- Compl. Buffer (Reorder Buff.)
- Complete
Embedded “Data Flow” Engine

- Read register or
- Assign register tag
- Advance instructions to reservation stations

- Monitor reg. tag
- Receive data being forwarded
- Issue when all operands ready

“Dynamic Execution”