CMSC 411 - A. Sussman (from D. O'Leary) 1

Computer Systems Architecture
CMSC 411
Unit 4 – Instruction-level parallelism

Alan Sussman
October 28, 2004

Administrivia

• Read Chapter 3 and Section A.8
  – in Ch. 3, only 3.1-3.8 and 3.10 required, rest is optional

What we already know about pipelining

• We need to avoid structural hazards, data hazards and control hazards in order to get optimal performance from the pipeline
  – Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls
• Accomplish this by techniques such as
  – instruction reordering
  – hardware modifications to detect branches earlier
  – compiler approaches to reduce branch delays
• Each technique reduces one or more of the stall components of the Pipeline CPI

What's new in Units 4 & 4b

• Some instructions can be executed independently of others. In fact, they could be executed in parallel if there was the hardware to do it
• Idea is to take advantage of this instruction level parallelism to do major rearrangements to how compilers generate code (Unit 4b) and how the MIPS (and other modern processors) pipeline executes code (Unit 4)

New techniques

• Hardware
  – dynamic pipeline scheduling
    • with scoreboard
    • with register renaming (Tomasulo)
  – dynamic branch prediction
  – issuing multiple instructions per cycle
  – speculation
  – dynamic memory disambiguation
• Software (compiler)
  – loop unrolling
  – compiler dependence analysis
  – software pipelining and trace scheduling
  – compiler speculation

Data dependences (background)

• If 2 instructions are parallel, can execute simultaneously in pipeline without stalls
  – assuming no structural hazards
• If 2 instructions are dependent, must be executed in order, but may sometimes be partially overlapped
Data dependences (cont.)

- Instruction \( j \) is **data dependent** on instruction \( i \) if either
  - instruction \( i \) produces a result that may be used by instruction \( j \), or
  - instruction \( j \) is data dependent on instruction \( k \), and instruction \( k \) is data dependent on instruction \( l \) (transitivity)

- Dependence implies a chain of one or more data hazards between the 2 instructions
  - potentially causing a pipelined processor to stall

- Dependences are properties of **programs**
  - and whether one causes a stall depends on the properties of the pipeline organization

Name dependences

- When 2 instructions use the same register or memory location (harder to detect), called a **name**, but no flow of data between them

- 2 types
  - *antidependence* between instruction \( i \) and instruction \( j \) when \( j \) writes a register or memory location that \( i \) reads
  - *output dependence* occurs when \( i \) and \( j \) write the same register or memory location

- In both cases, the instructions can execute at the same time, or be reordered, if the **name** is changed so the instructions don’t conflict
  - statically by compiler or dynamically by hardware
  - (usually only for registers — why?)

Control dependences

- To determine the ordering of an instruction, \( j \), with respect to a branch instruction so that it is executed in correct program order, and only when it should be
  - e.g., the statements in *then* part of an *if* statement are control dependent on the branch

Hazards

- True data dependences correspond to **RAW** data hazards
- Output dependences correspond to **WAW** hazards
- Antidependences correspond to **WAR** hazards

Control dependences (cont.)

- 2 constraints from control dependences
  - an instruction control dependent on a branch cannot be **moved before** the branch so that its execution is **no longer controlled** by the branch
  - an instruction not control dependent on a branch cannot be **moved after** the branch so that its execution is **controlled** by the branch

- But, can violate control dependences if don’t affect the correctness of the program
  - by preserving **exception behavior** and **data flow**
  - implemented by control hazard detection that causes control stalls, which can be reduced by various techniques (e.g., delayed branch)
Dynamic pipeline scheduling

• A hardware technique that can do a good job of scheduling, since it has perfect information about hazards
• Basic idea is to start each instruction as early as possible
• Means have to deal with instructions that complete out-of-order
• There is a lot to keep track of, and two main schemes to do the bookkeeping
  - scoreboard method
  - Tomasulo's method

Scoreboard method

• Origins: CDC 6600, late 1960s
• Scoreboard controls the progress of each instruction to ensure that hazards such as RAW, etc. are avoided
• ID basic pipeline stage split into 2 stages
  - Issue – decode instructions, check for structural hazards
  - Read operands – wait until no data hazards, then read operands
• Distinguish when instruction begins execution and when completes execution – in between it is in execution
• All instructions pass through issue stage in order, but can stall or bypass each other in second stage
  - so can get WAR hazards when instructions execute out of order
  - to have multiple instructions in EX stage simultaneously, use multiple functional units, or pipelined units, or both – equivalent for purposes of pipeline control

Computer Systems Architecture
CMSC 411
Unit 4 – Instruction-level parallelism

Alan Sussman
November 9, 2004

Scoreboard method

• Origins: CDC 6600, late 1960s
• Scoreboard controls the progress of each instruction to ensure that hazards such as RAW, etc. are avoided
• ID basic pipeline stage split into 2 stages
  - Issue – decode instructions, check for structural hazards
  - Read operands – wait until no data hazards, then read operands
• Distinguish when instruction begins execution and when completes execution – in between it is in execution
• All instructions pass through issue stage in order, but can stall or bypass each other in second stage
  - so can get WAR hazards when instructions execute out of order
  - to have multiple instructions in EX stage simultaneously, use multiple functional units, or pipelined units, or both – equivalent for purposes of pipeline control

Administivia

• HW #5 (Unit 4) out soon
• Project posted – due Dec. 3
  – Linux lab account info emailed – let us know if you didn’t get it
  – questions?
• Midterm returned Thursday (we hope)

Last time

• Data dependences
  – instruction \( i \) produces a result used, directly or transitively, by instruction \( j \)
  – true dependences in program may cause stalls from data (RAW) hazards
• Name dependences
  – 2 instructions use the same register (or memory location), but no data flows between them
  – antidependences (WAR hazards) and output dependences (WAW hazards)
  – changing name removes the dependence
    • take advantage of this in hardware techniques for exploiting ILP

Last time (cont.)

• Control dependences
  – maintain ordering of instructions with respect to conditional branches
  – can only violate them if preserve exception behavior and data flow
• Dynamic pipeline scheduling
  – idea is to start instructions as early as possible – as soon as input data is available
  – scoreboard
  – Tomasulo’s method
Scoreboard method

- Scoreboard controls the progress of each instruction to ensure that hazards such as RAW, etc. are avoided.
- ID basic pipeline stage split into 2 stages
  - Issue – decode instructions, check for structural hazards
  - Read operands – wait until no data hazards, then read operands
- Distinguish when instruction begins execution and when completes execution – in between it is in execution.
- All instructions pass through issue stage in order, but can stall or bypass each other in second stage
  - so can get WAR hazards when instructions execute out of order
  - to have multiple instructions in EX stage simultaneously, use multiple functional units, or pipelined units, or both – equivalent for purposes of pipeline control.

Scoreboard (cont.)

- One segment of the scoreboard keeps track of which pipe stage each instruction is in – Fig. A.52

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Read operands</th>
<th>Execution complete</th>
<th>Write result</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F6,F34(R2)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>L.D F2,F8(R3)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>MUL.D F0,F2,F4</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB.D F8,F6,F2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV.D F10,F6,F6</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F5,F8,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Functional units

- Processor has various functional units, perhaps
  - 1 ALU,
  - 2 multiply units,
  - 1 floating point adder,
  - and 1 divide unit
- Idea is to try to keep all of them busy

Scoreboard for functional units

<table>
<thead>
<tr>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Fi</th>
<th>Fj</th>
<th>Fk</th>
<th>Qj</th>
<th>Qk</th>
<th>RJ</th>
<th>RK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Yes</td>
<td>Load</td>
<td>F2</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>Yes</td>
<td>Mult</td>
<td>F0</td>
<td>F2</td>
<td>F4</td>
<td>Integer</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td>No</td>
<td>Sub</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Yes</td>
<td>Div</td>
<td>F10</td>
<td>F0</td>
<td>F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>Yes</td>
<td>Div</td>
<td>F10</td>
<td>F0</td>
<td>F6</td>
<td>Mult1</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Register status

- One segment of the scoreboard keeps track of which registers are expecting outputs from which functional units

<table>
<thead>
<tr>
<th>FU</th>
<th>Mult1</th>
<th>Integer</th>
<th>Add</th>
<th>Divide</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next, an example of the scoreboard method.
The bookkeeping

Each functional unit keeps track of:

- Unit: Opn, F1, F2, Q1, Q2.

For our convenience, we’ll also indicate:
- Inst, Stage.

And we’ll assume these initial register contents and flags:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0, F2, F4, F6, F8, F10, F12, F14</td>
<td>0, 3, 4, 6, 8, 10, 12, 14</td>
</tr>
<tr>
<td>F0, F2, F4, F6, F8, F10, F12, F14</td>
<td>0, 0, 0, 0, 0, 0, 0, 0</td>
</tr>
</tbody>
</table>

Scoreboard - bookkeeping

<table>
<thead>
<tr>
<th>Unit</th>
<th>Instruction</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mult1</td>
<td>ADD.D F0, F2, F4</td>
<td>Issue</td>
</tr>
<tr>
<td>Add1</td>
<td>ADD.d 3, 4, 0</td>
<td>Issue</td>
</tr>
<tr>
<td>S1</td>
<td>Issue</td>
<td></td>
</tr>
</tbody>
</table>

Time 0

S1: ADD.D F0, F2, F4

Assume 2 cycle add

Time 1

S2: MULT.D F2, F6, F8

Assume 4 cycle mult

Time 2

S3: MULT.D F10, F0, F2

Time 3

S4: ADD.D F0, F8, F6

Stalled - WAW hazard on F0
**Administrivia**

- HW #5 (Unit 4) out soon
- Project posted – due Dec. 3
  - questions?
- Midterm returned Tuesday

**Last time**

- Scoreboard
  - example in slides will be fixed and posted
  - instructions issue in-order, can complete out of order
  - hazard detection done at issue stage
    - forwarding only through registers, so need to avoid WAR, WAW hazards – stall if destination register is waiting for a needed result
    - RAW hazards dealt with by forwarding through register file
  - scoreboard keeps track of:
    - which pipe stage each instruction is in (issue, read operations, execute complete, writeback done)
    - for each functional unit – busy flag, operation to perform, source and destination registers, functional units supplying sources (if not from registers), ready flags for source registers
    - register status – which functional unit a result is coming from

**Limits to the Scoreboard method**

- Number and types of functional units
  - for structural hazards
- Number of entries in the scoreboard table (how many instructions can be issued simultaneously)
  - determines how far ahead pipeline can look for independent instructions (the window)
  - in our discussions, never goes beyond a branch
- Output dependences and antidependences
  - that lead to WAW and WAR stalls
- Parallelism available in the program
  - to find independent instructions to execute

**Tomasulo’s dynamic scheduling method**

- Origins: IBM 360/91, 3 years after CDC 6600
- Scoreboard controls the progress of each instruction and makes sure that hazards such as RAW, etc. are avoided
- In Tomasulo's method, these functions are decentralized in reservation stations

**Tomasulo’s method**

- Main differences with scoreboard method:
  - decentralized execution control
  - decentralized hazard detection
  - register renaming by having operands passed to reservation stations
    - eliminates WAW and WAR hazards
  - Only 3 kinds of pipeline stages, not 4:
    - **Issue**: send instruction to an empty reservation station, in FIFO order, along with any register contents it needs
      - also renames registers, removing WAR and WAW hazards
    - **Execute**: wait for all operands, check for RAW hazards, then execute the instruction
      - if multiple instructions ready in same cycle for same functional unit, f.p. units choose one arbitrarily
      - for loads/stores, compute effective address first, execute loads from load buffer as soon as memory unit available, for stores wait for value to be stored before send to memory unit
    - **Write result**: put result on the common data bus (CDB) to get it back to a register and to any other reservation stations that need it
      - stores write to memory here too

**Instruction status**

- Origins: IBM 360/91, 3 years after CDC 6600
- Scoreboard controls the progress of each instruction and makes sure that hazards such as RAW, etc. are avoided
- In Tomasulo's method, these functions are decentralized in reservation stations

**Tomasulo’s dynamic scheduling method**

- Origins: IBM 360/91, 3 years after CDC 6600
- Scoreboard controls the progress of each instruction and makes sure that hazards such as RAW, etc. are avoided
- In Tomasulo's method, these functions are decentralized in reservation stations

**Tomasulo’s method**

- Main differences with scoreboard method:
  - decentralized execution control
  - decentralized hazard detection
  - register renaming by having operands passed to reservation stations
    - eliminates WAW and WAR hazards

Instruction status (cont.)

• Note differences from scoreboard:
  – no check for WAW and WAR hazards, since renaming prevents them
  – CDB broadcasts results, so don't have to wait for registers to be filled
  – Load and Store are also treated as functional units

Advantages over scoreboard

• Distribution of hazard detection logic
  – from distributed reservation stations and CDB
  – can release multiple instructions at once from single result (if other operands available)

• Eliminates stalls for WAW and WAR hazards
  – from renaming registers using reservation stations
  – and from storing operands into reservation station as soon as they are available

Bookkeeping for Tomasulo method

• Each reservation station keeps track of
  – Busy flag
  – Op: the operation to be performed
  – Qj, Qk: sources of operands
    • already present
    • or to come from another reservation station (functional unit)
  – Vj, Vk: values of the operands
  – A: the memory address info for a load or store (eventually the effective address)

Bookkeeping (cont.)

• Register hardware keeps track of which reservation station will fill each register
  – Qi – number of reservation station containing the operation whose result will be stored into the register

• The load and store buffers have
  – busy flag
  – value to be stored
  – A: effective address

MIPS FP with Tomasulo’s algorithm

Instructions (Fig. 3.3)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Execute</th>
<th>Write Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F6,34(R2)</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>L.D F2,45(R3)</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL.D F0,F2,F4</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB.D F8,F2,F6</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV.D F10,F0,F6</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F6,F8,F2</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reservation Stations (Fig. 3.3)

Mult1Mem[34+R[R2]]DIVyesMult1
Load2R[F4]MULyesMult1
Add3
Load2Add1ADDyesAdd2
Load2Mem[34+R[R2]]SUbyesAdd1
45+R[R3]LoadyesLoad2
noLoad1

Register Status (Fig. 3.3)

An example of the Tomasulo method

The bookkeeping

Time 0
Assume 2-cycle add

The bookkeeping

Each functional unit keeps track of:

For our convenience, we'll also indicate:

Assume these initial register contents and flags:

And we'll assume these initial register contents and flags:

F4
F6
F8
F10
F12
F14
F2
F0
4
6
8
10
12
14
3
0
0
0
0
0
0
0
0
0
0
0
0
0
0

The bookkeeping

The bookkeeping

For our convenience, we'll also indicate:

Final
P0
P2
P4
P6
P8
P10
P12
P14

Name
BuVy
Op
Qj
Qk
Vi
Vj
Wj
Wk

Mult1
yes
Mult1
Mult2
Add1
Add2
Load1
Load2
Load1
Load2
Load1
Load2
Load1
Load2
Time 1

Assume 4 cycle mult.

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction</th>
<th>Mult1</th>
<th>Mult2</th>
<th>Add1</th>
<th>Add2</th>
<th>S1</th>
<th>Exec1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MULT.D F2, F6, F8</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>S2 Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Mult1 Mult.d 6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Mult2 x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Add1 Add.d 3 4 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Add2 x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F0 F2 F4 F6 F8 F10 F12 F14
0 3 4 6 8 10 12 14
a1 m1 0 0 0 0 0 0

Time 2

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction</th>
<th>Mult1</th>
<th>Mult2</th>
<th>Add1</th>
<th>Add2</th>
<th>S1</th>
<th>Exec2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MULT.D F10, F0, F2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>S3 Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Mult1 Mult.d 6</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Mult2 x x d1 x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Add1 Add.d 3 4 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Add2 x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F0 F2 F4 F6 F8 F10 F12 F14
0 3 4 6 8 10 12 14
a1 m1 0 0 0 0 0 0

Time 3

NOT STALLED - F0's contents already passed to all units that need it!

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction</th>
<th>Mult1</th>
<th>Mult2</th>
<th>Add1</th>
<th>Add2</th>
<th>S1</th>
<th>Exec3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD.D F0, F8, F6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>S4 Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Mult1 Mult.d 6</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Mult2 x x d1 x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Add1 Add.d 3 4 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Add2 Add.d 8 6 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To m2, but F0 not really necessary

F0 F2 F4 F6 F8 F10 F12 F14
7 3 4 6 8 10 12 14
a1 m1 0 0 0 0 m2 0 0

Time 4

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction</th>
<th>Mult1</th>
<th>Mult2</th>
<th>Add1</th>
<th>Add2</th>
<th>S1</th>
<th>Exec4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD.D F4, F6, F6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>S5 Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Mult1 Mult.d 6</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Mult2 x x d1 x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Add1 Add.d 6 6 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Add2 Add.d 8 6 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F0 F2 F4 F6 F8 F10 F12 F14
7 3 4 6 8 10 12 14
a1 m1 0 0 0 0 m2 0 0

Time 5

stalled

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction</th>
<th>Mult1</th>
<th>Mult2</th>
<th>Add1</th>
<th>Add2</th>
<th>S1</th>
<th>Exec5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD.D F2, F2, F6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>S6 Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Mult1 Mult.d 6</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Mult2 x x d1 x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Add1 Add.d 6 6 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Add2 Add.d 8 6 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F0 F2 F4 F6 F8 F10 F12 F14
7 3 4 6 8 10 12 14
a1 m1 0 0 0 0 m2 0 0

Time 6

stalled

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction</th>
<th>Mult1</th>
<th>Mult2</th>
<th>Add1</th>
<th>Add2</th>
<th>S1</th>
<th>Exec6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD.D F2, F2, F6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>S7 Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Mult1 Mult.d 6</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Mult2 x x d1 x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Add1 Add.d 6 6 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Add2 Add.d 8 6 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F0 F2 F4 F6 F8 F10 F12 F14
7 3 4 6 8 10 12 14
a1 m1 0 0 0 0 m2 0 0

stalled
### Loop-based example

- To show power of eliminating WAR and WAW hazards from dynamic register renaming

\[\text{Loop: } \text{L.D} \quad F0, 0(R1) \]
\[\text{MUL.D} \quad F4,F0,F2 \]
\[\text{S.D} \quad F4,0(R1) \]
\[\text{DADDUI} \quad R1,R1,-8 \]
\[\text{BNE} \quad R1,R2,\text{Loop} \]

- If predict branches are taken, multiple loop iterations can execute at same time – dynamic loop unrolling

### Example (Fig. 3.6)

- Assume all instructions issued in two successive iterations of loop, but none of f.p. loads/stores or operations has completed
  - don’t show integer ALU op, assume branch predicted as taken
- Once system reaches this state, 2 copies of loop could be sustained, with a CPI close to 1.0
  - if multiplies take less than 4 cycles to complete

### Instruction status (Fig. 3.6)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>From iteration</th>
<th>Issue</th>
<th>Execute</th>
<th>Write Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>MUL.D F4,F0,F2</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F0,0(R1)</td>
<td>2</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>MUL.D F4,F0,F2</td>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Register status (Fig. 3.6)

<table>
<thead>
<tr>
<th>Field</th>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
<th>F12</th>
<th>…</th>
<th>F30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qi</td>
<td>Load2</td>
<td>Multi2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

More on Loads/Stores

- Can be done in different order, if access different addresses
- If load and store access same address, either
  - load is before store in program order, so after interchange get WAR hazard
  - store is before load in program order, so after interchange get RAW hazard
- Interchanging 2 stores to same address results in WAW hazard

Loads / Stores (cont.)

- To decide whether a load can be executed at a given time, check if any uncompleted store (that precedes the load in program order) shares same memory address
  - similarly, for stores, wait until no earlier loads or stores share same address
- To find the address conflicts
  - for a load, after effective address computed, check against A field of all active store buffers
    - if a match, send to load buffer after conflicting store completes
  - for a store, check for conflicts in both load and store buffers

Computer Systems Architecture
CMSC 411
Unit 4 – Instruction-level parallelism

Alan Sussman
November 16, 2004

Administrivia

- HW #5 (Unit 4) posted – due date set soon
- Midterm returned today
  - median: 67  25%: 57  75%: 77
  - questions?
- Project due Dec. 3
  - questions?
- Quiz 3 postponed
  - until finish Unit 4, and turn in HW#5
  - date TBD

Last time

- Scoreboard
  - problems include WAW and WAR data hazards, structural hazards, size (window), available parallelism in program
- Tomasulo’s method
  - decentralize control to reservation stations
    - instructions to run + operands + flags
    - functional unit(s)
  - decentralized hazard detection
  - register renaming to eliminate WAW/WAR hazards
  - 3 pipeline stages
    - issue to empty reservation station, read operands, rename registers (use internal ones in reservation station)
    - execute – wait for operands (RAW hazards), execute instruction
    - write result to CDB, read into register and reservation stations
Dynamic branch prediction

- Dynamic scheduling techniques (scoreboard and Tomasulo algorithm) reduce data hazards
- Now: ways to reduce branch costs from control hazards

Branch-prediction buffers

- Simplest dynamic branch-prediction scheme
  - also called branch history table
- Records how frequently a branch has been taken in the past
- What is the buffer?
  - a small memory/cache area
- If there are more branches than locations in the buffer:
  - some locations are used for more than one branch instruction, which can reduce the accuracy of predictions
- How to find the right place in the buffer:
  - from the low order bits of the address of the branch instruction

Branch-prediction buffers (cont.)

- When to access the buffer:
  - fetch the contents during the ID pipeline cycle
  - update the contents after the branch is resolved
- What is stored in the buffer:
  - perhaps one bit per branch, indicating whether the branch was last taken or not
  - predict taken if it was last taken once.
  - otherwise predict not taken
Example: If the branch is for an inner loop, the prediction will be wrong on the 1st and last iterations, each time the loop is executed

Branch-prediction buffers (cont.)

- What is stored in the buffer (cont.):
  - perhaps a counter for each branch. If have 3 bits, for example, start the count at 4
  - add one every time the branch is taken
  - subtract one every time the branch is not taken
  - predict taken if the count is 4 or more
  - predict not taken if the count is 3 or less.
  - perhaps a correlating predictor that counts for this branch and for 1 or more earlier ones

How good are they? – Fig. 3.8

For a 4096 entry 2-bit prediction buffer

Correlating branch predictors

- To improve prediction accuracy, look at behavior of recent other branches than the one trying to predict
- Take advantage of correlation between behavior of different branches
  - also called two-level predictors
Simple example: MIPS code, with d in R1:

```
if (d == 0) BNEZ R1,L1 ; branch b1
if (d == 1) BNEZ R3,L2 ; branch b2
```

MIPS code, with d in R1:

```
if (d == 0) BNEZ R1,L1 ; branch b1
DADDIU R1,R0,#1
L1: DADDIU R3,R1,#-1
if (d == 1) BNEZ R3,L2 ; branch b2
... L2:
```
Example (cont.) – Fig. 3.10

<table>
<thead>
<tr>
<th>Initial value of d</th>
<th>d=0?</th>
<th>b1</th>
<th>Value of d before b2</th>
<th>d=1?</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>yes</td>
<td>not taken</td>
<td>1</td>
<td>yes</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>taken</td>
<td>1</td>
<td>yes</td>
<td>not taken</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>taken</td>
<td>2</td>
<td>no</td>
<td>taken</td>
</tr>
</tbody>
</table>

b1 not taken → b2 not taken

Example – 1 bit predictor

Predictor initialized to not taken – Fig. 3.11

<table>
<thead>
<tr>
<th>d=?</th>
<th>b1 prediction</th>
<th>b1 action</th>
<th>New b1 prediction</th>
<th>b2 prediction</th>
<th>b2 action</th>
<th>New b2 prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>

All branches mispredicted!

1-bit prediction with 1 bit correlation

Initialized to not taken/not taken – Fig. 3.13

<table>
<thead>
<tr>
<th>d=?</th>
<th>b1 prediction</th>
<th>b1 action</th>
<th>New b1 prediction</th>
<th>b2 prediction</th>
<th>b2 action</th>
<th>New b2 prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT/NT</td>
<td>T</td>
<td>T/NT</td>
<td>NT/NT</td>
<td>T</td>
<td>NT/NT</td>
</tr>
<tr>
<td>0</td>
<td>T/NT</td>
<td>NT</td>
<td>T/NT</td>
<td>NT</td>
<td>T</td>
<td>NT/NT</td>
</tr>
<tr>
<td>2</td>
<td>T/NT</td>
<td>T</td>
<td>T/NT</td>
<td>NT/NT</td>
<td>T</td>
<td>NT/NT</td>
</tr>
<tr>
<td>0</td>
<td>T/NT</td>
<td>NT</td>
<td>T/NT</td>
<td>NT/NT</td>
<td>T</td>
<td>NT/NT</td>
</tr>
</tbody>
</table>

Only misprediction is on first iteration/row

Comparing predictors

• Correlating vs. standard 2-bit scheme
  – using same number of state bits
• Number of state bits for \((m,n)\) predictor is:
  – \(2^m \times n \times 8\) prediction entries to select from with branch address
• 2-bit predictor with no global history is a \((0,2)\) predictor
  – \((0,2)\) predictor from Fig. 3.8 had 4K entries selected by branch address
  • total number of bits is \(2^2 \times 2 \times 4K = 8K\) bits
  – \((2,2)\) predictor from Fig. 3.14 has 64 entries, with 4 entries per branch address
    • total number of bits is \(2^2 \times 2 \times 16 = 128\) bits

Comparing 2-bit predictors

Correlating predictors (cont.)

• Predictor on last slide is a \((1,1)\) predictor
  – uses behavior of last branch to choose from among a pair of 1-bit branch predictors
• In general, an \((m,n)\) predictor uses behavior of last \(m\) branches to choose from \(2^m\) branch predictors
  – each is an \(n\)-bit predictor for a single branch
  – simple hardware to do this – can keep global history of most recent \(m\) branches in an \(m\)-bit shift register, and each bit records whether branch taken/not taken
  • and index branch prediction buffer using low-order bits of branch address with \(m\)-bit global history

Comparing 2-bit predictors

Fig. 3.15
Combining local and global predictors

- **Tournament predictors** use multiple predictors, usually one global and one local, combined with a selector
  - a form of multilevel branch prediction
  - selector is often just a 2-bit saturating counter per branch to choose from 2 predictors that is used with a finite state machine (see Fig. 3.16)
  - increment counter when "predicted" predictor is correct and other predictor is incorrect, decrement in opposite situation

Misprediction rates

Branch target buffers

- Another branch penalty reduction method
- BTB stores:
  - branch prediction information
  - predicted location of the instruction following the branch
- What is the buffer:
  - a small memory area, accessed with the address of the PC of the instruction fetched
- If there are more branches than locations in the buffer:
  - then some locations are used for more than one branch instruction, limiting the amount of information stored

Branch target buffers (cont.)

- When to access the buffer:
  - fetch the contents during the IF pipeline cycle
  - update the contents after the branch is resolved
- How to find the right entry in the buffer:
  - look for an entry that matches the PC
- What is stored in the buffer:
  - addresses of branch instructions
  - predicted target of each branch
  - branch prediction information
- Allows branch folding for unconditional branches (jumps)
  - zero cycle instruction!
  - by having BTB replace jump instruction in pipeline with branch target instruction returned from BTB

Branch target buffer – Fig. 3.19

Computer Systems Architecture
CMSC 411
Unit 4 – Instruction-level parallelism

Alan Sussman
November 18, 2004
Administivia

• Midterm questions?
• Project due Dec. 3
  – questions?
• Online course evaluation available at https://www.courses.umd.edu/online_evaluation

Last time

• Dynamic branch prediction
  – to reduce costs from control hazards
  – branch prediction buffer
    • cache to keep track of which way a branch went previously
    • use low-order bits of the instruction address to index into the cache
    • update an entry when branch direction is resolved
    • store 1 bit per branch (taken/not taken), or a counter
  – correlating predictors
    • to improve prediction accuracy, use behavior of recently executed branches (not just the one currently being predicted)
    • combined with local predictor, an \((m,n)\) predictor uses behavior of last \(m\) branches to choose from \(2^n\) predictors, each of which is an \(n\)-bit predictor for the current branch

Last time (cont.)

• Tournament branch predictor combines local (no correlation) and global (correlating) predictors with a selector, to choose the one that has been performing best recently
• Branch target buffer
  – store branch target (the address of the likely next instruction), in addition to prediction info, index with address of current instruction

Branch target buffer – Fig. 3.19

BTB algorithm – Fig. 3.20

Table: BTB penalties – Fig. 3.21

<table>
<thead>
<tr>
<th>Instruction in buffer</th>
<th>Prediction</th>
<th>Actual branch</th>
<th>Penalty cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>taken</td>
<td>taken</td>
<td>0</td>
</tr>
<tr>
<td>yes</td>
<td>taken</td>
<td>not taken</td>
<td>2</td>
</tr>
<tr>
<td>no</td>
<td>taken</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>no</td>
<td>not taken</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Issuing multiple instructions at one time

- Scoreboard and Tomasulo’s method reduce stalls from data hazards
- Branch buffers reduce stalls from control hazards
- Now talk about getting more parallelism by issuing several instructions at once
- Two currently used methods:
  - superscalar processors
  - VLIW (very long instruction word) processors

Superscalar MIPS

- Idea: issue several non-interfering instructions at each clock cycle
- The hardware has the burden of detecting interfering instructions
- Can be statically scheduled (using compiler techniques), or dynamically scheduled (using scoreboard or Tomasulo’s algorithm)
- For example, look at a superscalar MIPS processor that can issue two instructions at once:
  - one floating point instruction
  - one non-floating point (integer or branch) instruction

Example

\[
\text{\textbf{Loop:}} \\
\text{LD} \ F0, 0(R1) \quad x[i] = x[i] + s \\
\text{ADD.D} \ F4, F0, F2 \quad \text{uses F0 and F4} \\
\text{SD} \ F4(R1) \\
\text{LD} \ F6, 8(R1) \quad x[i-1] = x[i-1] + s \\
\text{ADD.D} \ F8, F6, F2 \quad \text{uses F6 and F8} \\
\text{SD} \ F8(R1) \\
\text{LD} \ F14, -16(R1) \quad x[i-2] = x[i-2] + s \\
\text{ADD.D} \ F14, F10, F2 \quad \text{uses F10 and F12} \\
\text{SD} \ F14, -24(R1) \quad x[i-3] = x[i-3] + s \\
\text{ADD.D} \ F16, F14, F2 \quad \text{uses F10 and F12} \\
\text{SD} \ F16, -32(R1) \quad \text{DSUBI R1, R1, #32 point to next element} \\
\text{BNEZ R1, Loop} \\
\]

5 approaches – Fig. 3.23

<table>
<thead>
<tr>
<th>Common name</th>
<th>Issue structure</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Interesting feature</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar (static)</td>
<td>dynamic</td>
<td>hardware</td>
<td>static</td>
<td>in-order execution</td>
<td>Sun UltraSparc II/III</td>
</tr>
<tr>
<td>Superscalar (dynamic)</td>
<td>dynamic</td>
<td>hardware</td>
<td>dynamic</td>
<td>some oo-oo execution</td>
<td>IBM Power2</td>
</tr>
<tr>
<td>Superscalar (speculative)</td>
<td>dynamic</td>
<td>hardware</td>
<td>dynamic, with speculation</td>
<td>Alpha 21264</td>
<td></td>
</tr>
<tr>
<td>VLIW/LIW</td>
<td>static</td>
<td>software</td>
<td>static</td>
<td>no hazards bet. packets</td>
<td>Trimedia, Intel i860</td>
</tr>
<tr>
<td>EPIC</td>
<td>static</td>
<td>mostly software</td>
<td>mostly</td>
<td>explicit dep. by compiler</td>
<td>Itanium</td>
</tr>
</tbody>
</table>

Example (cont.)

On single issue MIPS pipeline, takes 14 cycles

<table>
<thead>
<tr>
<th>Cycle</th>
<th>FP instruction</th>
<th>non-FP instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L.D \ F0, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L.D \ F6, 36(R1)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ADD.D \ F4, F0, F2</td>
<td>L.D \ F10, -16(R1)</td>
</tr>
<tr>
<td>4</td>
<td>ADD.D \ F8, F6, F2</td>
<td>L.D \ F14, -24(R1)</td>
</tr>
<tr>
<td>5</td>
<td>ADD.D \ F12, F10, F2</td>
<td>DSUBI R1, R1, -32</td>
</tr>
<tr>
<td>6</td>
<td>ADD.D \ F16, F14, F2</td>
<td>S.D \ F4, 32(R1)</td>
</tr>
<tr>
<td>7</td>
<td>S.D \ F8, 24(R1)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>S.D \ F12, 16(R1)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>BNEZ R1, Loop (delayed branch)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S.D \ F16, 8(R1)</td>
<td></td>
</tr>
</tbody>
</table>
Example (cont.)

- Need to use care not to exceed register traffic bounds
- It may be impossible to load an F register at the same time that the floating point unit is accessing/reading another F register – a structural hazard

VLIW MIPS

- Idea: have several non-interfering instructions stored in each instruction fetched
- Then the software (the compiler) has the burden of detecting interfering instructions
- Must be \textit{statically} scheduled by compiler

Example

- Assume a VLIW instruction can contain 2 memory references, 2 floating point operations, and 1 other operation each clock cycle
- Then the example loop takes 6 cycles, because latencies still limit performance

Example (cont.)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Memory</th>
<th>Memory</th>
<th>FP</th>
<th>FP</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L.D F0</td>
<td>L.D F6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L.D F10</td>
<td>L.D F14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ADD.D F4</td>
<td>ADD.D F8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ADD.D F12</td>
<td>ADD.D F16</td>
<td></td>
<td></td>
<td>DSUBI</td>
</tr>
<tr>
<td>5</td>
<td>S.D F4</td>
<td>S.D F8</td>
<td></td>
<td></td>
<td>BNEZ</td>
</tr>
<tr>
<td>6</td>
<td>S.D F12</td>
<td>S.D F16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Limitations on multiple issue

- Programs have limited parallelism and cannot keep all functional units busy – still have control hazards
- CPU hardware gets very complicated for issuing and executing
- Memory bandwidth requirements increase
- Superscalar: many decisions to be made very fast
- VLIW: code length increases, because of loop unrolling and many NOP instructions/operations
- Vector processors (Appendix G online) generally cheaper and faster.
- ... but recently, VLIW becoming more popular again (Intel/HP EPIC)

Hardware speculation

- A method for overcoming control dependences/hazards
- Branch prediction reduces stalls, but still want to execute instructions past a branch to get more instruction level parallelism, esp. with multiple issue
- \textit{Speculate} on the outcome of branches, and \textit{execute} the program as if the guesses are correct
- more than just dynamic scheduling plus branch prediction
- instructions are fetched, issued, and executed
- then fix things up if the speculation was incorrect
Hardware speculation (cont.)

- 3 key ideas
  - dynamic branch prediction, to choose which instructions to execute
  - speculation, to allow executing instructions before branches resolved (and effects undone if speculation was wrong)
  - dynamic scheduling, to deal with scheduling combinations of basic blocks
- Essentially a data flow execution
  - operations execute as soon as operands available
- We’ll use Tomasulo’s algorithm for dynamic scheduling, and just for FP unit

Extensions to Tomasulo’s algorithm

- Separate bypassing of results between instructions from actual completion of an instruction
  - so that an instruction can execute and produce results for other instructions, without performing any updates that can’t be undone – until know that the instruction should have been executed (control dependences resolved)
  - when instruction is no longer speculative:
    - update the register file, or memory
    - call this instruction commit

Implementing speculation

- Allow instructions to execute out-of-order, but make them commit in order
  - and prevent any action that can’t be undone (update state, take an exception, etc.) until instruction commits
- Requires some changes to standard pipeline sequence, and hardware buffers to hold results of instructions that have completed execution, but have not committed yet
  - a reorder buffer
  - also used to pass results between speculated instructions

Reorder buffer (ROB)

- Provides registers, like reservation stations
- Holds instruction result from time operation completes until instruction commits
- But no writeback to register file until instruction commits
  - so ROB supplies operands in this time interval
- Similar to store buffer in Tomasulo’s algorithm
  - in design shown, store buffer integrated into ROB

Reorder buffer (cont.)

- 4 fields in an ROB entry
  - instruction type – branch, store, register op (ALU or load)
  - destination – register number or memory address
  - value – result of instruction
  - ready – set when instruction completes execution, and value is ready

MIPS with Tomasulo + speculation

Fig. 3.29
Instruction execution – 4 steps

- **Issue**
  - Get instruction from instruction queue
  - Issue if both reservation station and empty ROB slot available
  - Send operands to reservation station if available in either registers or ROB
  - Send ROB entry number to reservation station for writing result
  - If reservations stations or ROB full, instruction stalls

Instruction execution (cont.)

- **Execute**
  - If an operand is not yet available, monitor CDB until it appears
  - When all operands available at reservation station, execute the operation
  - Instructions can take multiple cycles in this stage, loads require 2 steps (compute effective address and memory access), stores only need to compute effective address

Instruction execution (cont.)

- **Write result**
  - When result available, write on CDB (with ROB tag set in issue step), and from CDB to ROB, plus any reservation stations waiting
  - Mark reservation station available
  - For stores, if value to be stored is available, write into Value field of ROB entry for store
  - Otherwise, wait until it appears on CDB, then update Value field of ROB entry

Instruction execution (cont.)

- **Commit**
  - For “normal” case (instruction reaches head of ROB and its result is in the buffer), update the register with the result and remove instruction from ROB
  - For a store, update memory instead of result register
  - For an incorrectly predicted branch, the speculation was wrong – flush the ROB and reservation stations, and restart execution at correct branch target
  - Correctly predicted branch is just finished

Example – Fig. 3.30

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Address</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F6,34(R2)</td>
<td>F6</td>
<td>34</td>
<td>R2</td>
</tr>
<tr>
<td>L.D F2,45(R3)</td>
<td>F2</td>
<td>45</td>
<td>R3</td>
</tr>
<tr>
<td>MUL.D F0,F2,F4</td>
<td>F0</td>
<td>F2</td>
<td>F4</td>
</tr>
<tr>
<td>SUB.D F8,F6,F2</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
</tr>
<tr>
<td>DIV.D F10,F0,F6</td>
<td>F10</td>
<td>F0</td>
<td>F6</td>
</tr>
<tr>
<td>ADD.D F6,F8,F2</td>
<td>F6</td>
<td>F8</td>
<td>F2</td>
</tr>
</tbody>
</table>

- Assume 2 cycle add, 10 cycle multiply, 40 cycle divide
- Status tables will show when MUL.D is ready to commit

Example (cont.)

<table>
<thead>
<tr>
<th>Reservation stations</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
<th>Dest</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load1</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load2</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add1</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add2</td>
<td>no</td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>Mul1</td>
<td>no</td>
<td>MUL.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mul2</td>
<td>yes</td>
<td>DIV.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reservation stations

- Mem[45 + R[R3]]
- R[F4]
- Mem[34 + R[R2]]
- #3
- #5
Example (cont.)

Reorder buffer

<table>
<thead>
<tr>
<th>Entry</th>
<th>Busy</th>
<th>Instruction</th>
<th>State</th>
<th>Dest.</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no</td>
<td>L.D</td>
<td>Commit</td>
<td>F6</td>
<td>Mem[34-R[R2]]</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>L.D</td>
<td>Commit</td>
<td>F2</td>
<td>Mem[45-R[R3]]</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>MUL.D</td>
<td>Write Result</td>
<td>F0</td>
<td>R2 × R[F4]</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>SUB.D</td>
<td>Write Result</td>
<td>F8</td>
<td>#1 - #2</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>DIV.D</td>
<td>Execute</td>
<td>F10</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td>ADD.D</td>
<td>Write Result</td>
<td>F6</td>
<td>#4 + #2</td>
</tr>
</tbody>
</table>

FP register status

<table>
<thead>
<tr>
<th>Field</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>F10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROB</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busy</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Computer Systems Architecture
CMSC 411
Unit 4 – Instruction-level parallelism

Alan Sussman
November 23, 2004

Last time

- Multiple instruction issue
  - superscalar – static, dynamic, speculative
  - hazard detection in hardware
  - VLIW – and EPIC
  - hazard detection in software (compiler)

- Superscalar MIPS
  - fetch then issue 0, 1, or 2 instructions to correct functional unit
  - execute them as with Tomasulo (or scoreboard)

- Speculation
  - fetch, issue and execute instructions past branches, fixing up if speculation was incorrect
  - don’t commit instruction until know whether it should be executed, and
  - commit instructions in order, even though they can execute out-of-order
  - use a reorder buffer – also passes results between speculated instructions

Speculation - Exceptions

- Processor with ROB can provide precise exceptions/interrupts
  - don’t take exception caused by program until commit
  - at that point, flush any pending instructions
  - works because commit happens in program order

- Much harder to do precise exceptions with basic Tomasulo algorithm
  - instructions can complete before earlier issued instructions, with register or memory writes that can’t be undone – imprecise exceptions
  - makes some kinds of exceptions (e.g., page faults) hard to handle

Administrivia

- HW #5 due Nov. 30
- Project due Dec. 3
  - questions?
- Quiz 3 scheduled for Dec. 7
  - quiz 4 cancelled, re-weight other quizzes in raw score
- For Unit 6, read Chapter 7
  - except 7.8, 7.12-13
- Online course evaluation available at https://www.courses.umd.edu/online_evaluation
Loop example – Fig. 3.31

Loop:
L.D F0,0(R1)
MUL.D F4,F0,F2
S.D F4,0(R1)
DADDIU R1,R1,#-8
BNE R1,R2,Loop

• Assume all instructions in loop issued twice
• L.D and MUL.D from iteration 1 committed, all other instructions finished executing
• Effective address for stores already computed, and have left ROB

Loop example (cont.)

Reorder buffer

<table>
<thead>
<tr>
<th>Entry</th>
<th>Busy</th>
<th>Inst.</th>
<th>State</th>
<th>Dest.</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no</td>
<td>L.D</td>
<td>Commit</td>
<td>F0</td>
<td>Mem[0+R[R1]]</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>MUL.D</td>
<td>Commit</td>
<td>F4</td>
<td>#1 × R[F2]</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>S.D</td>
<td>Write result</td>
<td>0+R[R1]</td>
<td>#2</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>DADDUI</td>
<td>Write result</td>
<td>R1</td>
<td>R[R1] – 8</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>BNE</td>
<td>Write result</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td>L.D</td>
<td>Write result</td>
<td>F0</td>
<td>Mem[#4]</td>
</tr>
<tr>
<td>7</td>
<td>yes</td>
<td>MUL.D</td>
<td>Write result</td>
<td>F4</td>
<td>#6 × R[F2]</td>
</tr>
<tr>
<td>8</td>
<td>yes</td>
<td>S.D</td>
<td>Write result</td>
<td>0+04</td>
<td>#7</td>
</tr>
<tr>
<td>9</td>
<td>yes</td>
<td>DADDUI</td>
<td>Write result</td>
<td>R1</td>
<td>#4 – 8</td>
</tr>
<tr>
<td>10</td>
<td>yes</td>
<td>BNE</td>
<td>Write result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Speculation with multiple issue

• Similar to extending Tomasulo algorithm for multiple issue
  process multiple instructions per clock, assign reservation stations and reorder buffer entries
• Major challenges are instruction issue and monitoring CDBs for instruction completion
  – and must be able to handle multiple instruction commits per cycle
• As example in book shows (Figs. 3.33 and 3.34), speculation helps when there are data dependent branches that limit performance
  – requires good branch prediction
  – incorrect speculation could hurt performance

Design considerations

• Register renaming
  – an alternative to ROB
  – employ a larger set of physical registers than the ones visible in the architecture (R and F)
  – replace the ones in the reservations stations and ROB
  – renaming happens at instruction issue
  – renaming dest register removes WAW and WAR hazards
  – speculation recovery easy since physical register holding instruction dest doesn’t become architectural register until commit
  – hard part is knowing when deallocation is OK
  • when no longer corresponds to a physical register, and nothing will use it

How much to speculate

• How much to speculate
  – cost comes from incorrect speculation of an expensive event (e.g., a 2nd level cache miss)
  – solution is to only allow low-cost exceptional events in speculative mode (e.g., 1st level cache miss)
  – processor waits until instruction is not speculative before handling expensive exceptional event
Design considerations (cont.)

- Speculating through multiple branches
  - when is this beneficial?
    - very high branch frequencies
    - clustering of branches
    - long delays in functional units
  - really just makes recovery more complicated
  - IBM Power2 can even predict and speculate on more than 1 branch per cycle!

Performance study: Limitations of ILP

- Question is how much ILP is available in real programs
  - limits the amount of parallelism, no matter how much hardware is used
  - tells you how much can enhance performance from architectural decisions, on top of increases in circuit speeds
- Results that follow are mostly from a study by Wall in 1993 on SPEC92 benchmarks

Hardware model used

- Start from an ideal processor
  - all artificial constraints on ILP removed
  - only limits on ILP are from actual data flows through registers or memory
- Assumptions
  1. register renaming – infinite number of virtual registers available (no WAW or WAR hazards) and unbounded number of instructions issued simultaneously
  2. branch prediction – all perfectly predicted
  3. jump prediction – all perfectly predicted – combined with 2, same as perfect speculation and unbounded window
  4. memory address alias analysis – all addresses know exactly, so load can be moved before store if addresses not identical
- 2 and 3 eliminate all control dependences
- 1 and 4 eliminate all but true data dependences

Hardware model (cont.)

- Unlimited instruction issue, with no restrictions on lookahead or on types of instructions issued
- All functional units have 1 cycle latency
- Perfect caches, so all loads/stores complete in 1 cycle
- This processor is pretty much unrealizable, but is where we start
First restriction study

- Limit window size and maximum issue count
  - Window size is the number of instructions looked at for simultaneous execution (the number in-flight)
  - Current window sizes max out around 64-128
    - But real functional units pipelined, not single cycle
    - And real processors need window to hold outstanding memory references (cache misses)

Restricting window size – Fig. 3.36

Second restriction study

- Realistic branch and jump prediction
  - Fix window size at 2K and issue at most 64 instructions per clock
  - 5 levels of branch prediction
    - Perfect – for branches and jumps
    - Tournament-based – correlating 2-bit, noncorrelating 2-bit, and selector to choose best one for each branch, 48K bits total, 97% accurate on SPEC92, plus essentially perfect jump predictor
    - Standard 2-bit with 512 2-bit entries, plus 16 entry procedure return predictor
    - Static – based on profile history of program
    - None – only predict jumps – parallelism only within basic block

Effect of branch prediction

Again, f.p. programs show more available parallelism

Third restriction study

- Effects of finite registers
  - Huge tournament predictor (150K bits) plus large jump and return predictors
  - Now reduce number of registers for renaming from infinite in ideal processor – start from a base of 32 GP and 32 FP

Branch prediction accuracy

Accuracy better for f.p. programs, and when prediction not very accurate, misprediction penalty is very high
Fourth restriction study

- Imperfect alias analysis
  - 3 other memory models, instead of perfect
    - global/stack perfect – and assumes all heap references conflict (idealized version of best a compiler could do currently)
    - inspection – compile time interference checks, just looking for obvious memory conflicts
    - none – all memory references conflict
  - first 2 same for Fortran programs

Imperfect alias analysis

Example: P6 Microarchitecture

P6 microarchitecture

- Dynamically scheduled
- Translates each IA-32 instruction into series of micro-operations (uops) to be executed in pipeline – each uop is like a MIPS instruction
- Up to 3 IA-32 instructions fetched, decoded, and translated per cycle, max of 6 uops generated per cycle
- Pipeline is out-of-order and speculative, with register renaming and a reorder buffer (ROB)
Pipeline stalls

• Lead to processor not committing as many instructions as possible in a cycle
  – fewer than 3 insts fetched – inst cache miss
  – fewer than 3 insts issued – too many uops
  – not all uops generated can issue – lack of reservation stations or ROB entries
  – data dependences everywhere – reservation stations and ROBs
  – data cache miss – all res. stations and ROBs waiting
  – branch mispredicts – pipeline flushes and refills

Decode stalls – Fig. 3.51

resource capacity is the big problem

Data cache misses – Fig. 3.53

hard to read, but L2 misses are the problem

Fallacies

• Processors with lower CPIs will always be faster
• Processors with faster clock rates will always be faster
• Sophisticated pipelines with slower clocks sometimes win from using more ILP

uop completions per cycle

Fig. 3.56

shows benefit of a dyn. scheduled pipeline

Pitfalls

• Emphasizing improved CPI by increasing issue rate, but sacrificing clock rate, can lead to lower performance
  – complexity introduced to improve CPI can slow clock rate
• Improving only one part of a multiple-issue processor may not improve overall performance
  – need to find the bottleneck (i.e. Amdahl’s law)
• Sometimes bigger and dumber is better
  – different applications can produce different behaviors