EECS 252 Graduate Computer Architecture

Lec 5 – Out-of-Order Completion

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~culler
http://www-inst.eecs.berkeley.edu/~cs252

Review

• Data stationary pipeline control
  – Micro-instruction & PC track down the pipe
  – Accumulate state
• Implementing bubbles, stalls, forwarding, multicyle operations
• Branch prediction
  – Static vs dynamic
  – N-bit saturating counters
  – Local and global history
  – Correlated predictors, Tournament, GSHARE
  – Branch target buffers, return address predictors

Outline

• Relax pipeline design to allow out-of-order completions
  – Cray-1: register reservations
• Relax pipeline to allow out-of-order issue
  – CDC 6600: Scoreboard
• Compiler optimizations for ILP
• Superscalar issue
• Maybe Go back and finish exceptions

Pipelining with Reg. Reservations

• Assumptions
  1. Multiple pipelined function units of different latency
  – able to accept operations at issue rate
  – may be exceptions (e.g., divide)
  2. Issue instructions in order
  3. Operand fetch in order
  4. Completion out of order
  – short ops may bypass long ones
  5. Some shared resources (e.g., reg write port)
• Implications
  – WAR hazard still resolved by pipeline flow (2 & 3)
  – RAW, WAW, and structural still present
• Design philosophy (ala Cray)
  – Resolve hazards as instruction is issued into pipeline
  – Pipeline is non-blocking

Resolving Structural Hazards

With static pipeline flow, resource usage is known in advance
• Instruction requires X at t ticks after issue
• If reservationX[t] is clear, issue instr and set bit
• Otherwise, delay till clear
• At each tick the reservationX[] shifts by one, so will eventually clear
• Multiple resources? Range of delays?

Basic Issue Model

• Issue unit checks for all hazards
  – Structural RAW, WAW
• Holds issue while hazards exist
• Upon issue, register values provided to F.U
• Executes to completion without blocking

NOW Handout Page 1
### Hazard Resolution

- **Structural**
  - Op code => resource usage
  - Check resource resv
  - Set on issue

- **Data**
  - Add reservation bit one each register
  - Check RegRsv for source and destination registers
  - Hold issue till clear
  - Set bit on destination register
  - Clear bit on dest reg. Write

- **Questions:**
  - Forwarding?

### Example

```
Add r1 := r2 + r3
Add r2 := r2 + 4
Lod r5 := mem[r1+16]
Lod r6 := mem[r1+32]
Mul r7 := r5 * r6
Bnz r1, foo
Sub r7 := r0 - r0
```

### Cray-1 Discussion

- **Technological Assumptions**
- Why no forwarding?
- Longevity of the ISA?
- Instruction cache?
  - Four blocks (RR) of 16x4 "parcels"
  - Issue delayed on miss
  - 2 CP for change of block
- Branch delays?
  - Brach op code delayed till second parcel is obtained
  - 5 clocks (reg zero, nz, pos, neg)
- I/O system?

### Pipelining with Scoreboarding

- **Assumptions**
  1. Multiple function units of different latency
     - Especially non-pipelined units
  2. Issue instructions whenever FU available, unless would cause multiple outstanding writes to same register
     - Operand fetch out of order
     - Completion out of order
  3. Some shared resources (e.g., reg write port)
- **Implications**
  - Need to resolve RAW, WAR, WAW and structural
- **Design philosophy (ala CDC 6600)**
  - Issue unit tracks all outstanding dependences
  - Holds issue if structural or WAW hazard
  - Informs FUs when hazards resolved
  - FUs fetch operands from register file and proceed

### Scoreboard Operation

- **Issue**
  - Hold while FU unavailable or destination register reserved (by FU !)

- **Read operands**
  - SB informs FU with all sources available to fetch & go
  - Limited by read ports

- **Write back**
  - SB schedules one FU to write
  - Waits no FU waiting to fetch (old version) of reg

### Example

```
Add r1 := r2 + r3
Add r2 := r2 + 4
Lod r5 := mem[r1+16]
Lod r6 := mem[r1+32]
Mul r7 := r5 * r6
Bnz r1, foo
Sub r7 := r0 - r0
```
Discussion

• Technological Assumptions
• Extend to allow forwarding?
• How do loads and stores work?
• Instruction cache?
• I/O system?

Case Study: MIPS R4000 (200 MHz)

• 8 Stage Pipeline:
  – IF—first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  – IS—second half of access to instruction cache.
  – RF—instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  – EX—execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  – DF—data fetch, first half of access to data cache.
  – DS—second half of access to data cache.
  – TC—tag check, determine whether the data cache access hit.
  – WB—write back for loads and register-register operations.

• 8 Stages: What is impact on Load delay? Branch delay? Why?

Case Study: MIPS R4000

• FP Adder, FP Multiplier, FP Divider
• Last step of FP Multiplier/Divider uses FP Adder HW
• 8 kinds of stages in FP units:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Functional unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>FP adder</td>
<td>Mantissa ADD stage</td>
</tr>
<tr>
<td>D</td>
<td>FP divider</td>
<td>Divide pipeline stage</td>
</tr>
<tr>
<td>E</td>
<td>FP multiplier</td>
<td>Exception test stage</td>
</tr>
<tr>
<td>M</td>
<td>FP multiplier</td>
<td>First stage of multiplier</td>
</tr>
<tr>
<td>N</td>
<td>FP multiplier</td>
<td>Second stage of multiplier</td>
</tr>
<tr>
<td>R</td>
<td>FP adder</td>
<td>Rounding stage</td>
</tr>
<tr>
<td>S</td>
<td>FP adder</td>
<td>Operand shift stage</td>
</tr>
<tr>
<td>U</td>
<td>Unpack FP numbers</td>
<td></td>
</tr>
</tbody>
</table>

MIPS R4000 Floating Point

<table>
<thead>
<tr>
<th>FP Instr</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Subtract</td>
<td>U</td>
<td>A</td>
<td>A+R</td>
<td>R+S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td>U</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>N</td>
<td>N+R</td>
<td>N+R</td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>U</td>
<td>A</td>
<td>R</td>
<td>D^A</td>
<td>...</td>
<td>D+R</td>
<td>D+R</td>
<td>D+R</td>
<td>A</td>
</tr>
<tr>
<td>Square root</td>
<td>U</td>
<td>E</td>
<td>(A+R)^1/2</td>
<td>...</td>
<td>A</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td>U</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute value</td>
<td>U</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP compare</td>
<td>U</td>
<td>A</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS FP Pipe Stages

<table>
<thead>
<tr>
<th>Stages:</th>
<th>M</th>
<th>N</th>
<th>R</th>
<th>S</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>First stage of multiplier</td>
<td>A Mantissa ADD stage</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Second stage of multiplier</td>
<td>D Divide pipeline stage</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rounding stage</td>
<td>E Exception test stage</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operand shift stage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unpack FP numbers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R4000 Performance

- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfulfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)
Advanced Pipelining and Instruction Level Parallelism (ILP)

- ILP: Overlap execution of unrelated instructions
- gcc 17% control transfer
  - 5 instructions + 1 branch
  - Beyond single block to get more instruction level parallelism
- Loop level parallelism one opportunity
  - First SW, then HW approaches
- DLX Floating Point as example
  - Measurements suggest R4000 performance FP execution has room for improvement

Can we make CPI closer to 1?

- Let’s assume full pipelining:
  - If we have a 4-cycle latency, then we need 3 instructions between a producing instruction and its use:
    - `mult $f0,$f2,$f4`
    - `delay-1`
    - `delay-2`
    - `add $f6,$f10,$f0`

   ![Earliest forwarding for 4-cycle instructions]

   ![Earliest forwarding for 1-cycle instructions]

FP Loop: Where are the Hazards?

Loop: `LD F0,0(R1) ; F0=vector element`
- `ADD F4,F0,F2 ; add scalar from F2`
- `SD 0(R1),F4 ; store result`
- `SUBI R1,R1,8 ; decrement pointer 8B (DW)`
- `BNEZ R1,Loop ; branch R1!=zero`

Instruction Instruction Latency in producing result using result clock cycles
<table>
<thead>
<tr>
<th>ALU op</th>
<th>ALU op</th>
<th>Store double</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>FP</td>
<td>op</td>
<td>2</td>
</tr>
<tr>
<td>Load</td>
<td>FP</td>
<td>op</td>
<td>0</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>op</td>
<td>0</td>
</tr>
</tbody>
</table>

- Where are the stalls?

FP Loop Showing Stalls

1. Loop: `LD F0,0(R1); F0=vector element`
2. `stall`
3. `ADD F4,F0,F2 ; add scalar from F2`
4. `stall`
5. `SD 0(R1),F4 ; store result`
6. `SUBI R1,R1,8 ; decrement pointer 8B (DW)`
7. `BNEZ R1,Loop ; branch R1!=zero`
8. `stall`

Instruction Instruction Latency in producing result using result clock cycles
<table>
<thead>
<tr>
<th>ALU op</th>
<th>ALU op</th>
<th>Store double</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>FP</td>
<td>op</td>
<td>2</td>
</tr>
<tr>
<td>Load</td>
<td>FP</td>
<td>op</td>
<td>0</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>op</td>
<td>0</td>
</tr>
</tbody>
</table>

- 9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

1. Loop: `LD F0,0(R1)`
2. `stall`
3. `ADD F4,F0,F2`
4. `SUBI R1,R1,8`
5. `BNEZ R1,Loop ; delayed branch`
6. `SD 8(R1),F4 ; altered when move past SUBI`

Instruction Instruction Latency in producing result using result clock cycles
<table>
<thead>
<tr>
<th>ALU op</th>
<th>ALU op</th>
<th>Store double</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>FP</td>
<td>op</td>
<td>3</td>
</tr>
<tr>
<td>Load</td>
<td>FP</td>
<td>op</td>
<td>1</td>
</tr>
</tbody>
</table>

Swap BNEZ and SD by changing address of SD

Unroll Loop Four Times (straightforward way)

1. Loop: `LD F0,0(R1)`
2. `ADD F4,F0,F2`
3. `SD 0(R1),F4 ; drop SUBI & BNEE`
4. `LD F6,-8(R1)`
5. `ADD F8,F6,F2`
6. `SD -8(R1),F8 ; drop SUBI & BNEE`
7. `LD F10,-16(R1)`
8. `ADD F12,F10,F2`
9. `SD -16(R1),F12 ; drop SUBI & BNEE`
10. `LD F14,-24(R1)`
11. `ADD F16,F14,F2`
12. `SD -24(R1),F16`
13. `SUBI R1,R1,16 ; alter to 4*8`
14. `BNEZ R1,Loop`
15. `NOP`

6 clocks: Unroll loop 4 times code to make faster?

Unroll loop to minimize stalls?

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration
Assumes R1 is multiple of 4
Unrolled Loop That Minimizes Stalls

1. Loop: LD F0,0(R1)
2. LD F6,-8(R1)
3. LD F10,-16(R1)
4. LD F14,-24(R1)
5. ADDD F4,F0,F2
6. ADDD F8,F6,F2
7. ADDD F12,F10,F2
8. ADDD F16,F14,F2
9. SD 0(R1),F4
10. SD -8(R1),F8
11. SD -16(R1),F12
12. SD -24(R1),F16
13. SUBI R1,R1,#32
14. BNEZ R1,LOOP
15. SD -32(R1),F20

- What assumptions made when moved code?
  - OK to move store past SUBI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar DLX: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Pipe Stages

- Int. instruction IF ID EX MEM WB
- FP instruction IF ID EX MEM WB

Type

- Int. instruction IF ID EX MEM WB
- FP instruction IF ID EX MEM WB

- 1 cycle load delay expands to 3 instructions in SS
  - Instruction in right half can’t use it, nor instructions in next slot

Loop Unrolling in Superscalar

<table>
<thead>
<tr>
<th>Integer instruction</th>
<th>FP instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R0(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F6,R0(R1)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>ADDD F0,F0,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>LD F14,-24(R1)</td>
<td>ADDD F4,F2,F2</td>
<td>4</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>ADDD F12,F10,F2</td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>ADDD F16,F14,F2</td>
<td>6</td>
</tr>
<tr>
<td>SD -8(R1),F8</td>
<td>ADDD F20,F18,F2</td>
<td>7</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>SD -24(R1),F16</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>SUBI R1,R1,#40</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>BNEZ R1,LOOP</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration (1.5X)

SuperScalar Issue Rules

- Datapath has specific kinds of functional parallelism
- Fetch packet of instructions
- “Issue rules”: constraints over and beyond dependencies
  - Ex: one arithmetic or branch, one load/store, one FP

VLIW: Very Large Instruction Word

- Each “instruction” has explicit coding for multiple operations
  - In EPIC, grouping called a “packet”
  - In Transmeta, grouping called a “molecule” (with “atoms” as ops)
  - In 1976 (same year as Cray-1) Floating Point Systems AP120B
    *poor mans Cray*, 2 MFLOPS for 50k vs 20 MFLOPS for 12M

- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
  - 16 to 24 bits per field => 716 or 112 bits to 724 or 168 bits wide

- Need compiling technique that schedules across several branches

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency
Note: Need more registers in VLIW (15 vs. 6 in SS)
Summary

- Increasingly powerful (and complex) dynamic mechanism for detecting and resolving hazards
  - In-order pipeline, in-order op-fetch with register reservations, in-order issue with scoreboard
  - Weaken the timing and flow assumptions
  - Allow later instructions to proceed around ones that are stalled
  - Facilitate multiple issue
  - Not quite powerful enough to unroll loops dynamically
    » Stop when attempt to rebind a new value to a reg.
- Compiler techniques make it easier for HW to find the ILP
  - Reduces the impact of more sophisticated organization
  - Requires a larger architected namespace
  - Easier for more structured code