CPE 631 Session 19
Exploiting ILP with SW Approaches

Electrical and Computer Engineering
University of Alabama in Huntsville

Outline
- Review:
  - Basic Pipeline Scheduling and Loop Unrolling
  - Multiple Issue: Superscalar, VLIW
- Software Pipelining
- Multiple Issue with Dynamic Scheduling

Basic Pipeline Scheduling: Example
- Simple loop:
  ```
  for(i=1; i<=1000; i++)
  x[i]=x[i-1] + a;
  ```
- Assumptions:
  - Instruction producing result
  - Instruction using result
  - Latency in clock cycles
  - FP ALU op
  - Another FP ALU op
  - 3
  - FP ALU op
  - Store double
  - 2
  - Load double
  - FP ALU op
  - 1
  - Load double
  - Store double
  - 0

Revised FP loop to minimise stalls
- 6 clocks per iteration (1 stall); but only 3 instructions do the actual work processing the array (LD, ADD, SD)
- => Unroll loop 4 times to improve potential for instr. scheduling

Swap BNEZ and SD by changing address of SD SUBI is moved up

6 clocks per iteration (1 stall); but only 3 instructions do the actual work processing the array (LD, ADD, SD)
=> Unroll loop 4 times to improve potential for instr. scheduling
Unrolled Loop

This loop will run 28 cc (14 stalls) per iteration; each LD has one stall, each ADDD 2, SUBI 1, BNEZ 1, plus 14 instruction issue cycles - or 28/4=7 for each element of the array (even slower than the scheduled version)!

Rewrite loop to minimize stalls

Unrolled Loop that Minimise Stalls

This loop will run 14 cycles (no stalls) per iteration - or 14/4=3.5 for each element!

Assumptions that make this possible:
- move LDs before SDs
- move SD after SUBI and BNEZ
- use different registers

When is it safe for compiler to do such changes?

Superscalar MIPS

- Superscalar MIPS: 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

Time [clocks]

Note: FP operations extend EX cycle

Loop Unrolling in Superscalar

Unrolled 5 times to avoid delays

This loop will run 12 cycles (no stalls) per iteration - or 12/5=2.4 for each element of the array
The VLIW Approach

- VLIWs use multiple independent functional units
- VLIWs package the multiple operations into one very long instruction
- Compiler is responsible to choose instructions to be issued simultaneously

```
I  F  ID  E  W
I+1 F  E  ID  W
Instr.
```

Time [clocks]

Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Mem. Ref. 1</th>
<th>Mem. Ref. 2</th>
<th>FP1</th>
<th>FP2</th>
<th>Int/Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD F4,F0,F2</td>
<td>ADD F4,F0,F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD F6,F0,F2</td>
<td>ADD F6,F0,F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD F8,F6,F2</td>
<td>ADD F8,F6,F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD F10,F6,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD F12,F10,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD F14,F12,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBUI R1,R1,#24</td>
<td>ADD F0,F0,F0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in SW)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SD 0(R1),F4</td>
</tr>
<tr>
<td>1</td>
<td>ADDD F4,F0,F2</td>
</tr>
<tr>
<td>2</td>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>3</td>
<td>SUBUI R1,R1,#24</td>
</tr>
<tr>
<td>4</td>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>5</td>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>6</td>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>7</td>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>8</td>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>9</td>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>10</td>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>11</td>
<td>BNEZ R1,LOOP</td>
</tr>
</tbody>
</table>

Software Pipelining Example

<table>
<thead>
<tr>
<th>Before: Unrolled 3 times</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
</tr>
<tr>
<td>ADDD F4,F0,F2</td>
</tr>
<tr>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>LD F6,8(R1)</td>
</tr>
<tr>
<td>ADDD F8,F6,F2</td>
</tr>
<tr>
<td>SD 8(R1),F6</td>
</tr>
<tr>
<td>LD F10,16(R1)</td>
</tr>
<tr>
<td>ADDD F12,F10,F2</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
</tr>
<tr>
<td>SUBUI R1,R1,#24</td>
</tr>
<tr>
<td>BNEZ R1,LOOP</td>
</tr>
</tbody>
</table>

After: Software Pipelined

<table>
<thead>
<tr>
<th>After: Software Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>SD 0(R1),F0</td>
</tr>
<tr>
<td>ADD F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>SD 0(R1),F0</td>
</tr>
</tbody>
</table>

5 cycles per iteration
E.g., four-issue static superscalar
- 4 instructions make one issue packet
- Fetch examines each instruction in the packet in the program order
  - instruction cannot be issued will cause a structural or data hazard
  - due to an instruction earlier in the issue packet or
due to an instruction already in execution
- can issue from 0 to 4 instruction per clock cycle

Multiple Issue with Dynamic Scheduling

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

Assumptions:
One FP and one integer operation can be issued;
Resources: ALU (int + effective address), a separate pipelined FP for each operation type, branch prediction hardware, 1 CDB
2 cc for loads, 3 cc for FP Add
Branches single issue, branch prediction is perfect
Multiple Issue with Dynamic Scheduling:
Resource Usage

- DADDIU waits for ALU used by S.D
- Add one ALU dedicated to effective address calculation
- Use 2 CDBs
- Draw table for the dual-issue version of Tomasulo’s pipeline

Multiple Issue with Dynamic Scheduling:

### CDB #1
<table>
<thead>
<tr>
<th>Line</th>
<th>ALU</th>
<th>FPU</th>
<th>Cache</th>
<th>CDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

### CDB #2
<table>
<thead>
<tr>
<th>Line</th>
<th>ALU</th>
<th>FPU</th>
<th>Cache</th>
<th>CDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>S.D</td>
<td>L.D</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

## Tomasulo’s Pipeline

### Execution
- **Wait for BNE111097**
  - LD.D F0,0(R1)
  - ADD.D F4,F0,F2
  - S.D 0(R1), F4
  - DADDIU R1,R1,-#8
  - BNE R1,R2,Loop
- **Wait for BNE8764**
  - LD.D1294
  - ADD.D1375
  - S.D 0(R1), F4
  - DADDIU R1,R1,-#8
  - BNE R1,R2,Loop

### Issue
- **Wait for DAIDU53**
  - BNE R1,R2,Loop
  - DADDIU R1,R1,-#8

### Write
- **Wait for DAIDU432**
  - DADDIU R1,R1,-#8
  - S.D 0(R1), F4
  - LD.D851
  - ADD.D F4,F0,F2

### Access
- **Wait for DAIDU321**
  - LD.D F0,0(R1)
  - Com.Write

### Comm.
- **Wait for DAIDU243**
  - DADDIU R1,R1,-#8
  - S.D 0(R1), F4

### Data Cache
- **Wait for DAIDU156**
  - DADDIU R1,R1,-#8
  - S.D 0(R1), F4
What about Precise Interrupts?

- State of machine looks as if no instruction beyond faulting instructions has issued.

- Tomasulo had:
  - In-order issue, out-of-order execution, and out-of-order completion.
  - Need to “fix” the out-of-order completion aspect so that we can find precise breakpoint in instruction stream.

Relationship between precise interrupts and speculation

- Speculation: guess and check
- Important for branch prediction:
  - Need to “take our best shot” at predicting branch direction.
  - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly:
    - This is exactly same as precise exceptions!
  - Technique for both precise interrupts/exceptions and speculation: in-order completion or commit

HW support for precise interrupts

- Need HW buffer for results of uncommitted instructions:
  - reorder buffer
    - 3 fields: instr, destination, value
    - Use reorder buffer number instead of reservation station when execution completes
    - Supplies operands between execution complete & commit
    - (Reorder buffer can be operand source => more registers like RS)
  - Instructions commit
    - Once instruction commits, result is put into register
    - As a result, easy to undo speculated instructions on mispredicted branches or exceptions

Four Steps of Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   - If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called “dispatch”)

2. Execution—operate on operands (EX)
   - When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called “issue”)

3. Write result—finish execution (WB)
   - Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. Commit—update register with reorder result
   - When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called “graduation”)
What are the hardware complexities with reorder buffer (ROB)?

How do you find the latest version of a register?

- (As specified by Smith paper) need associative comparison network
- Could use future file or just use the register result status buffer to track which specific reorder buffer has received the value

Need as many ports on ROB as register file