Recall: Achieving Precise Exceptions

- **Use pipeline to sort this out!**
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don’t act on exception until it reach WB stage
- **Handle interrupts through “faulting noop” in IF stage**
- **When instruction reaches end of MEM stage:**
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!

Examples of stalls/bubbles

- **Exceptions: Flush everything above**
  - Prevent instructions following exception from committing state
  - Freeze fetch until exception resolved
- **Stalls: Introduce brief stalls into pipeline**
  - Decode stage recognizes that current instruction cannot proceed
  - Freeze fetch stage
  - Introduce “bubble” into EX stage (instead of forwarding stalled inst)
  - Can stall until condition is resolved
  - Examples:
    - mfhi, mflo: need to wait for multiply/divide unit to finish
    - “Break” instruction for Lab5: stall until release line received
    - Load delay slot handled this way as well
Recall: 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?

Recall: Case Study: MIPS R4000

<table>
<thead>
<tr>
<th>TWO Cycle</th>
<th>IF</th>
<th>IS</th>
<th>RF</th>
<th>EX</th>
<th>DF</th>
<th>TC</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Latency</td>
<td>IF</td>
<td>IS</td>
<td>RF</td>
<td>EX</td>
<td>DF</td>
<td>DS</td>
<td>TC</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>IS</td>
<td>RF</td>
<td>EX</td>
<td>DF</td>
<td>DS</td>
<td>TC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>THREE Cycle</th>
<th>IF</th>
<th>IS</th>
<th>RF</th>
<th>EX</th>
<th>DF</th>
<th>DS</th>
<th>TC</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Latency</td>
<td>IF</td>
<td>IS</td>
<td>RF</td>
<td>EX</td>
<td>DF</td>
<td>DS</td>
<td>TC</td>
<td>WB</td>
</tr>
<tr>
<td>(conditions evaluated during EX phase)</td>
<td>IF</td>
<td>IS</td>
<td>RF</td>
<td>EX</td>
<td>DF</td>
<td>DS</td>
<td>TC</td>
<td>WB</td>
</tr>
</tbody>
</table>

Delay slot plus two stalls
Branch likely cancels delay slot if not taken

Can we somehow make CPI closer to 1?

\* Let’s assume full pipelining:
  - If we have a 4-cycle instruction, then we need 3 instructions between a producing instruction and its use:

```
multf $F0,$F2,$F4
delay-1
delay-2
delay-3
addf $F6,$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
NOP ;delayed branch slot

Instruction | Instruction | Latency in clock cycles
producing result | using result |
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
Integer op | Integer op | 0

\* Where are the stalls?
FP Loop Showing Stalls

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

9 clocks: Rewrite code to minimize stalls?

Unroll Loop Four Times (straightforward way)

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

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration
Assumes R1 is multiple of 4
CPI = 27/15 = 1.8

Revise FP Loop Minimizing Stalls

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

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

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 SUBI R1,R1,#32
13 BNEZ R1,LOOP
14 SD 8(R1),F16

14 clock cycles, or 3.5 per iteration
CPI = 14/14 = 1

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?

When safe to move instructions?

3/31/03 ©UCB Spring 2003 CS152 / Kubiatowicz Lec15.11
Two main variations: Superscalar and VLIW

Superscalar: varying no. instructions/cycle (1 to 6)
- Parallelism and dependencies determined/resolved by HW
- IBM PowerPC 604, Sun UltraSparc, DEC Alpha 21164, HP 7100

Very Long Instruction Words (VLIW): fixed number of instructions (16) parallelism determined by compiler
- Pipeline is exposed; compiler must schedule delays to get right result

Explicit Parallel Instruction Computer (EPIC)/ Intel
- 128 bit packets containing 3 instructions (can execute sequentially)
- Can link 128 bit packets together to allow more parallelism
- Compiler determines parallelism, HW checks dependencies and forwards/stalls

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

Type PipeStages
Int. instruction IF ID EX MEM WB
FP instruction IF ID EX MEM WB
Int. instruction IF ID EX MEM WB
FP instruction IF ID EX MEM WB
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

Limits of Superscalar

While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
- Exactly 50% FP operations
- No hazards

If more instructions issue at same time, greater difficulty of decode and issue
- Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue

VLIW: 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 can execute in parallel
- E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
  - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
- Need compiling technique that schedules across several branches

Loop Unrolling in Superscalar

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

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>LD F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F10,-4(R1)</td>
<td>LD F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F4, F0, F2</td>
<td>ADDD F8,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td></td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F16,F14,F2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>LD F30,-64(R1)</td>
<td></td>
<td>ADDD F20,F18,F2</td>
<td>ADDD F24,F22,F2</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>ADDD F26,F24,F26</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>ADDD F28,F26,F26</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td>ADDD F14,F14,F14</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>SD -40(R1),F24</td>
<td>ADDD F12,F12,F12</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration

Need more registers in VLIW (EPIC => 128int + 128FP)

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)

Software Pipelining Example

Before: Unrolled 3 times
1) LD F0,0(R1)
2) ADDD F4,F0,F2
3) SD 0(R1),F4
4) LD F6,-8(R1)
5) ADDD F8,F6,F2
6) SD -8(R1),F8
7) LD F10,-16(R1)
8) ADDD F12,F10,F2
9) SD -16(R1),F12
10) SUBI R1,R1,#24
11) BNEZ R1,LOOP

After: Software Pipelined
1) SD 0(R1),F4 ; Stores M[i]
2) ADDD F4,F0,F2 ; Adds to M[i-1]
3) LD F0,-16(R1) ; Loads M[i-2]
4) SUBI R1,R1,#8
5) BNEZ R1,LOOP

Software Pipelining with Loop Unrolling in VLIW

- Symbolic Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling

Software pipelined across 9 iterations of original loop
- In each iteration of above loop, we:
  - Store to m,m-8,m-16 (iterations I-3,I-2,I-1)
  - Compute for m-24,m-32,m-40 (iterations I,I+1,I+2)
  - Load from m-48,m-56,m-64 (iterations I+3,I+4,I+5)

- 9 results in 9 cycles, or 1 clock per iteration
- Average: 3.3 ops per clock, 66% efficiency

Note: Need less registers for software pipelining
(only using 7 registers here, was using 15)
**Administrivia**

- Should be working on Lab 5 by now!
  - Remember: a *Working* processor is necessary for full credit...
  - Due Tonight: Problem 0 (Lab evaluation of Lab 4)
  - Due Tomorrow night: Division of labor for Lab 5
- Boards? Hopefully... This will be an historic lab (if nothing else)
  - Problems are bound to occur
  - Should be able to completely demonstrate your processor in simulation
  - Top level module is not quite correct at moment
  - Memory may give us trouble
  - Debugging should be interesting
    - Memory-mapped I/O: input from switches, output to HEX LEDs
    - Use Break instruction to help debug: will stop with code on LEDs
- Get started early!!!
- More info on advanced topics:
  - *Computer Architecture: A Quantitative Approach* by John Hennessy and David Patterson
  - Should be on reserve at Engineering library

**Can we use HW to get CPI closer to 1?**

- Why in HW at run time?
  - Works when can’t know real dependence at compile time
  - Compiler simpler
  - Code for one machine runs well on another
- Key idea: Allow instructions behind stall to proceed:
  - DIVD F0, F2, F4
  - ADDD F10, F0, F8
  - SUBD F12, F8, F14
- Out-of-order execution ⇒ out-of-order completion.
- Disadvantages?
  - Complexity
  - Precise interrupts harder! (Talk about this next time)

**Problems?**

- How do we prevent WAR and WAW hazards?
- How do we deal with variable latency?
  - Forwarding for RAW hazards harder.

**The Big Picture: Where are We Now?**

- The Five Classic Components of a Computer
  - Processor
  - Control
  - Memory
  - Datapath
  - Input
  - Output

- Today’s Topics:
  - Recap last lecture
  - Hardware loop unrolling with Tomasulo algorithm
  - Administrivia
  - Speculation, branch prediction
  - Reorder buffers
Scoreboard: a bookkeeping technique

- Out-of-order execution divides ID stage:
  1. Issue—decode instructions, check for structural hazards
  2. Read operands—wait until no data hazards, then read operands

- Scoreboards date to CDC6600 in 1963

- Instructions execute whenever not dependent on previous instructions and no hazards.

- CDC 6600: In order issue, out-of-order execution, out-of-order commit (or completion)
  - No forwarding!
  - Imprecise interrupt/exception model for now

Scoreboard Implications

- Out-of-order completion => WAR, WAW hazards?

- Solutions for WAR:
  - Stall writeback until registers have been read
  - Read registers only during Read Operands stage

- Solution for WAW:
  - Detect hazard and stall issue of new instruction until other instruction completes

- No register renaming!

- Need to have multiple instructions in execution phase => multiple execution units or pipelined execution units

- Scoreboard keeps track of dependencies between instructions that have already issued.

- Scoreboard replaces ID, EX, WB with 4 stages

Scoreboard Architecture (CDC 6600)

Four Stages of Scoreboard Control

- Issue—decode instructions & check for structural hazards (ID1)
  - Instructions issued in program order (for hazard checking)
  - Don’t issue if structural hazard
  - Don’t issue if instruction is output dependent on any previously issued but uncompleted instruction (no WAW hazards)

- Read operands—wait until no data hazards, then read operands (ID2)
  - All real dependencies (RAW hazards) resolved in this stage, since we wait for instructions to write back data.
  - No forwarding of data in this model!
Four Stages of Scoreboard Control

° **Execution**—operate on operands (EX)
  - The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.

° **Write result**—finish execution (WB)
  - Stall until no WAR hazards with previous instructions:

    - Example:
      ```
      DIVD F0, F2, F4
      ADDD F10, F0, F8
      SUBD F8, F8, F14
      ```

    - CDC 6600 scoreboard would stall SUBD until ADDD reads operands

Three Parts of the Scoreboard

° **Instruction status:**
  - Which of 4 steps the instruction is in

° **Functional unit status:**—Indicates the state of the functional unit (FU). 9 fields for each functional unit
  - Busy: Indicates whether the unit is busy or not
  - Op: Operation to perform in the unit (e.g., + or –)
  - Fi: Destination register
  - Fj,Fk: Source-register numbers
  - Qj,Qk: Functional units producing source registers Fj, Fk
  - Rj,Rk: Flags indicating when Fj, Fk are ready

° **Register result status**—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register

CDC 6600 Scoreboard

° Speedup 1.7 from compiler; 2.5 by hand
  - BUT slow memory (no cache) limits benefit

° Limitations of 6600 scoreboard:
  - No forwarding hardware
  - Limited to instructions in basic block (small window)
  - Small number of functional units (structural hazards), especially integer/load store units
  - Do not issue on structural hazards
  - Wait for WAR hazards
  - Prevent WAW hazards

Summary #1/2: Compiler techniques for parallelism

° **Loop unrolling** ⇒ Multiple iterations of loop in software:
  - Amortizes loop overhead over several iterations
  - Gives more opportunity for scheduling around stalls

° **Software Pipelining** ⇒ Take one instruction from each of several iterations of the loop
  - Software overlapping of loop iterations
  - Today will show hardware overlapping of loop iterations

° **Very Long Instruction Word machines (VLIW)** ⇒
  - Multiple operations coded in single, long instruction
    - Requires sophisticated compiler to decide which operations can be done in parallel
    - Trace scheduling ⇒ find common path and schedule code as if branches didn’t exist (+ add “fixup code”)

° All of these require additional registers
Summary #2/2

- HW exploiting ILP
  - Works when can’t know dependence at compile time.
  - Code for one machine runs well on another

- Key idea of Scoreboard: Allow instructions behind stall to proceed (Decode => Issue instr & read operands)
  - Enables out-of-order execution => out-of-order completion
  - ID stage checked both for structural & data dependencies
  - Original version didn’t handle forwarding.
  - No automatic register renaming