Review: Three Advantages of Virtual Memory

- **Translation:**
  - Program can be given consistent view of memory, even though physical memory is scrambled.
  - Makes multithreading reasonable (now used a lot).
  - Only the most important part of program (“Working Set”) must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.

- **Protection:**
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior (Read Only, Invisible to user programs, etc).
  - Kernel data protected from User programs.
  - Very important for protection from malicious programs.

- **Sharing:**
  - Can map same physical page to multiple users (“Shared memory”)

Review: Translation Look-Aside Buffers

- **Translation Look-Aside Buffers (TLB)**
  - Cache on translations.
  - Fully Associative, Set Associative, or Direct Mapped.

- **TLBs are:**
  - Small – typically not more than 128 – 256 entries.
  - Fully Associative.

Exceptions: Traps and Interrupts

(Hardware)
Example: Device Interrupt
(Say, arrival of network message)

```
add    r1, r2, r3
subi   r4, r1, #4
slli   r4, r4, #2

Hiccup(!)
```

```
lw     r2, 0(r4)
lw     r3, 4(r4)
add     r2, r2, r3
sw      8(r4), r2
```

```
```

Raise priority
Reenable All Ints
Save registers
```

```
```

PC saved
Disable Ints
Supervisor Mode
```

```
```

Check if multiple interrupts
Clear Int
```

```
```

PC restored
```

```
```

Restore registers
Clear current Int
Disable All Ints
Restore priority
RTE
```

```
```

```
```

```
```

“Interrupt Handler”
```

```
```

```
```

```
```

```
```

Alternative: Polling
(again, for arrival of network message)

```
Disable Network Intr

... subi r4, r1, #4
     slli r4, r4, #2
     lw     r1, 20(r0)
     lw     r2, 0(r1)
     addi   r3, r0, #5
     sw     0(r1), r3

lw     r1, 12(r0)
beq    r1, no_mess
lw     r1, 20(r0)
lw     r2, 0(r1)
addi   r3, r0, #5
sw     0(r1), r3
```

```
```

```
```

```
```

“Polling Point
(check device register)
```

```
```

```
```

“Handler”
```

```
```

Polling is faster/slower than Interrupts.

- Polling is faster than interrupts because
  - Compiler knows which registers in use at polling point. Hence, do not need to save and restore registers (or not as many).
  - Other interrupt overhead avoided (pipeline flush, trap priorities, etc).
- Polling is slower than interrupts because
  - Overhead of polling instructions is incurred regardless of whether or not handler is run. This could add to inner-loop delay.
  - Device may have to wait for service for a long time.
- When to use one or the other?
  - Multi-axis tradeoff
    - Frequent/regular events good for polling, as long as device can be controlled at user level.
    - Interrupts good for infrequent/irregular events
    - Interrupts good for ensuring regular/predictable service of events.
  - Traps: relevant to the current process
    - Faults, arithmetic traps, and synchronous traps
    - Invoke software on behalf of the currently executing process
  - Interrupts: caused by asynchronous, outside events
    - I/O devices requiring service (DISK, network)
    - Clock interrupts (real time scheduling)
  - Machine Checks: caused by serious hardware failure
    - Not always restartable
    - Indicate that bad things have happened.
      - Non-recoverable ECC error
      - Machine room fire
      - Power outage

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

2/1/2010 CS252-S10, Lecture 4 5
2/1/2010 CS252-S10, Lecture 4 6
2/1/2010 CS252-S10, Lecture 4 7
2/1/2010 CS252-S10, Lecture 4 8
A related classification: Synchronous vs. Asynchronous

- **Synchronous**: means related to the instruction stream, i.e., during the execution of an instruction
  - Must stop an instruction that is currently executing
  - Page fault on load or store instruction
  - Arithmetic exception
  - Software Trap Instructions

- **Asynchronous**: means unrelated to the instruction stream, i.e., caused by an outside event.
  - Does not have to disrupt instructions that are already executing
  - Interrupts are asynchronous
  - Machine checks are asynchronous

- **SemiSynchronous (or high-availability interrupts)**:
  - Caused by external event but may have to disrupt current instructions in order to guarantee service

---

**Interrupt Priorities Must be Handled**

```
add r1, r2, r3
subi r4, r1, #4
slli r4, r4, #2
Hiccup(!)
lw r2, 0(r4)   lw r1, 20(r0)
lw r3, 4(r4)   lw r2, 0(r1)
add r2, r2, r3 addi r3, r0, #5
sw 8(r4), r2 sw 0(r1), r3
...              ...
```

Note that priority must be raised to avoid recursive interrupts!

---

**Interrupt Controller**

- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can’t be disabled

**Interrupt controller hardware and mask levels**

- Operating system constructs a hierarchy of masks that reflects some form of interrupt priority.
- For instance:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Software interrupts</td>
</tr>
<tr>
<td>2</td>
<td>Network Interrupts</td>
</tr>
<tr>
<td>4</td>
<td>Sound card</td>
</tr>
<tr>
<td>5</td>
<td>Disk Interrupt</td>
</tr>
<tr>
<td>6</td>
<td>Real Time clock</td>
</tr>
<tr>
<td>∞</td>
<td>Non-Maskable Ints (power)</td>
</tr>
</tbody>
</table>

- This reflects the an order of urgency to interrupts
- For instance, this ordering says that disk events can interrupt the interrupt handlers for network interrupts.
Can we have fast interrupts?

- Pipeline Drain: Can be very Expensive
- Priority Manipulations
- Register Save/Restore
  - 128 registers + cache misses + etc.

SPARC (and RISC I) had register windows
- On interrupt or procedure call, simply switch to a different set of registers
- Really saves on interrupt overhead
  - Interrupts can happen at any point in the execution, so compiler cannot help with knowledge of live registers.
  - Conservative handlers must save all registers
  - Short handlers might be able to save only a few, but this analysis is complicated
- Not as big a deal with procedure calls
  - Original statement by Patterson was that Berkeley didn’t have a compiler team, so they used a hardware solution
  - Good compilers can allocate registers across procedure boundaries
  - Good compilers know what registers are live at any one time
- However, register windows have returned!
  - IA64 has them
  - Many other processors have shadow registers for interrupts

Supervisor State
- Typically, processors have some amount of state that user programs are not allowed to touch.
  - Page mapping hardware/TLB
    - TLB prevents one user from accessing memory of another
    - TLB protection prevents user from modifying mappings
  - Interrupt controllers -- User code prevented from crashing machine by disabling interrupts. Ignoring device interrupts, etc.
  - Real-time clock interrupts ensure that users cannot lockup/crash machine even if they run code that goes into a loop:
    - “Preemptive Multitasking” vs “non-preemptive multitasking”
- Access to hardware devices restricted
  - Prevents malicious user from stealing network packets
  - Prevents user from writing over disk blocks
- Distinction made with at least two-levels: USER/SYSTEM (one hardware mode-bit)
  - x86 architectures actually provide 4 different levels, only two usually used by OS (or only 1 in older Microsoft OSs)

Entry into Supervisor Mode
- Entry into supervisor mode typically happens on interrupts, exceptions, and special trap instructions.
- Entry goes through kernel instructions:
  - Interrupts, exceptions, and trap instructions change to supervisor mode, then jump (indirectly) through table of instructions in kernel
    - intvec: j handle_int0
    - ... j handle_trap0
    - OS “System Calls” are just trap instructions:
      - read(fd, buffer, count) => st 20(r0), r1
      - ... st 28(r0), r3
      - trap $READ
- OS overhead can be serious concern for achieving fast interrupt behavior.
CS 252 Administrivia

- Textbook Reading for next few lectures:
  - Computer Architecture: A Quantitative Approach, Chapter 2
- Reading assignment for this week are up. Will work to get much further ahead.
  - Should be using web form to submit your summaries
  - Remember: I’ve given you 5 slip days on paper summaries
  - Technically, these paper summaries are due before class (1:00)
- Go to handouts page:
  - All readings put up so far are there
- You should try to do the prerequisite exams (look at handouts page)
  - See if you have good enough understanding of prerequisite material

Precise Interrupts/Exceptions

- An interrupt or exception is considered precise if there is a single instruction (or interrupt point) for which:
  - All instructions before that have committed their state
  - No following instructions (including the interrupting instruction) have modified any state.
- This means, that you can restart execution at the interrupt point and “get the right answer”
  - Implicit in our previous example of a device interrupt:
    » Interrupt point is at first lw instruction

Precise interrupt point may require multiple PCs

- On SPARC, interrupt hardware produces “pc” and “npc” (next pc)
- On MIPS, only “pc” – must fix point in software

Why are precise interrupts desirable?

- Many types of interrupts/exceptions need to be restartable. Easier to figure out what actually happened:
  - I.e. TLB faults. Need to fix translation, then restart load/store
  - IEEE gradual underflow, illegal operation, etc:
    e.g. Suppose you are computing: \( f(x) = \sin(x) \)
    Then, for \( x \rightarrow 0 \)
    \[ f(0) = \frac{0}{0} \Rightarrow NaN \text{ illegal operation} \]
    Want to take exception, replace NaN with 1, then restart.
- Restartability doesn’t require preciseness. However, preciseness makes it a lot easier to restart.
- Simplify the task of the operating system a lot
  - Less state needs to be saved away if unloading process.
  - Quick to restart (making for fast interrupts)
Precise Exceptions in simple 5-stage pipeline:

- Exceptions may occur at different stages in pipeline (i.e. out of order):
  - Arithmetic exceptions occur in execution stage
  - TLB faults can occur in instruction fetch or memory stage
- What about interrupts? The doctor’s mandate of “do no harm” applies here: try to interrupt the pipeline as little as possible
- All of this solved by tagging instructions in pipeline as “cause exception or not” and wait until end of memory stage to flag exception
  - Interrupts become marked NOPs (like bubbles) that are placed into pipeline instead of an instruction.
  - Assume that interrupt condition persists in case NOP flushed
  - Clever instruction fetch might start fetching instructions from interrupt vector, but this is complicated by need for supervisor mode switch, saving of one or more PCs, etc

Another look at the exception problem

- 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 reache WB stage
- Handle interrupts through “faulting noop” in IF stage
- When instruction reaches WB stage:
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!

Precise interrupts when instructions executing in arbitrary order?

- Jim Smith’s classic paper (you will read this) discusses several methods for getting precise interrupts:
  - In-order instruction completion
  - Reorder buffer
  - History buffer
- We will discuss these after we see the advantages of out-of-order execution.

Impact of Hazards on Performance
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.

- **Questions:**
  - How are stores handled?
  - Why TC stage for data but not instructions?
  - What is impact on Load delay? Branch delay? Why?

Delayed Stores for Write Pipeline

- Store Data placed into buffer until checked by TC stage
- Written to cache when bandwidth available
  - i.e. Written during following store’s DF/DS slots or earlier
  - Must be checked against future loads until placed into cache
- Aside: Why TC stage, but no similar stage for Inst Cache?
  - Instruction Cache Tag check done during decode stage (overlapped)
  - Must check Data Cache tag before writeback! (can’t be undone)

MIPS R4000 Floating Point

- 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>FP adder</td>
<td>Unpack FP numbers</td>
</tr>
</tbody>
</table>

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>DS</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>
<td>WB</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>
<td>WB</td>
</tr>
</tbody>
</table>

| THREE Cycle | IF | IS | RF | EX | DF | DS | TC | WB |
| Branch Latency | IF | IS | RF | EX | DF | DS | TC | WB |
|               | IF | IS | RF | EX | DF | DS | TC | WB |

Delay slot plus two stalls
Branch likely cancels delay slot if not taken
MIPS FP Pipe Stages

<table>
<thead>
<tr>
<th>FPInstr</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>S+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>E+M</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>N+A</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>U</td>
<td>A</td>
<td>R</td>
<td>D2^8</td>
<td>...</td>
<td>D+A</td>
<td>D+R</td>
<td>D+R</td>
<td>D+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>

**Stages:**
- **M**: First stage of multiplier
- **N**: Second stage of multiplier
- **R**: Rounding stage
- **S**: Operand shift stage
- **U**: Unpack FP numbers

R4000 Pipeline: Branch Behavior

<table>
<thead>
<tr>
<th>Clock Number</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>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Branch inst</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></td>
<td>Delay Slot</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></td>
<td>Branch Inst+8</td>
<td>IF</td>
<td>IS</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td>Branch Inst+12</td>
<td>IF</td>
<td>IS</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td>Branch Targ</td>
<td>IF</td>
<td>IS</td>
<td>RF</td>
<td>EX</td>
<td>DF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- On a **taken branch**, there is a one cycle delay slot, followed by two lost cycles (nullified insts).
- On a **non-taken branch**, there is simply a delay slot (following two cycles not lost).
- This is bad for loops. Could benefit from Branch Prediction (later lecture)

R4000 Performance

- Not ideal CPI of 1:
  - **Load stalls** (1 or 2 clock cycles)
  - **Branch stalls** (2 cycles + unfilled 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 suggests 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:

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>0</td>
</tr>
</tbody>
</table>
```

- Earliest forwarding for 4-cycle instructions:
  - Fetch: multf $F0, $F2, $F4
  - Decode: delay-1
  - Ex1: delay-2
  - Ex2: delay-3
  - Ex3: addf $F6, $F10, $F0

- Earliest forwarding for 1-cycle instructions:
  - Fetch: multfd
  - Decode: delay1
  - Ex1: delay2
  - Ex2: delay3
  - WB: multf

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

- 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 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

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

Swap BNEZ and SD by changing address of SD

- 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

Swap BNEZ and SD by changing address of SD

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

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 ; 8-32 = -24

14 clock cycles, or 3.5 per iteration

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?

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Pipe Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int. instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

- 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>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F0,0(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F6,-8(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>2</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>ADDD F8,F6,F2</td>
<td>3</td>
</tr>
<tr>
<td>LD F14,-24(R1)</td>
<td>ADDD F12,F10,F2</td>
<td>4</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>ADDD F16,F14,F2</td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>ADDD F20,F18,F2</td>
<td>6</td>
</tr>
<tr>
<td>SD -8(R1),F8</td>
<td>ADDD F16,F14,F2</td>
<td>7</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>ADDD F20,F18,F2</td>
<td>8</td>
</tr>
<tr>
<td>SD -24(R1),F16</td>
<td>ADDD F20,F18,F2</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)
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)
- 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 => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>2</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>ADDD F4,F0,F2</td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td>ADDD F12,F10,F2</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>SD -8(R1),F8</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>SD -24(R1),F16</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td>SD -40(R1),F24</td>
</tr>
<tr>
<td>SD -60(R1),F28</td>
<td>BNEZ R1,LOOP</td>
</tr>
</tbody>
</table>

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)

Another possibility: 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 Diagram](chart)

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

- 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

5 cycles per iteration
Software Pipelining with Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP operation 2</th>
<th>Int. op/ Clock branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,-48(R1)</td>
<td>ST 0(R1),F4</td>
<td>ADDD F4,F0,F2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F6,-56(R1)</td>
<td>ST -8(R1),F8</td>
<td>ADDD F8,F6,F2</td>
<td>SUBI R1,R1,#24</td>
<td>2</td>
</tr>
<tr>
<td>LD F10,-40(R1)</td>
<td>ST 8(R1),F12</td>
<td>ADDD F12,F10,F2</td>
<td>BNEZ R1,LOOP</td>
<td>3</td>
</tr>
</tbody>
</table>

- 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)

Compiler Perspectives on Code Movement

- Compiler concerned about dependencies in program
- Whether or not a HW hazard depends on pipeline
- Try to schedule to avoid hazards that cause performance losses
- (True) Data dependencies (RAW if a hazard for HW)
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- If dependent, can’t execute in parallel
- Easy to determine for registers (fixed names)
- Hard for memory (“memory disambiguation” problem):
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?

Where are the data dependencies?

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

Compiler Perspectives on Code Movement

- Another kind of dependence called name dependence: two instructions use same name (register or memory location) but don’t exchange data
- Antidependence (WAR if a hazard for HW)
  - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
- Output dependence (WAW if a hazard for HW)
  - Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.
Where are the name dependencies?

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

How can remove them?

From different loop iterations, does 20(R6) = 20(R6)?

There were no dependencies between some loads and stores so they could be moved by each other.

Compiler Perspectives on Code Movement

- Name Dependencies are Hard to discover for Memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Our example required compiler to know that if R1 doesn't change then:

  0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)

Called “register renaming”

- Final kind of dependence called control dependence
- Example

  if p1 {S1;};
  if p2 {S2;};

  S1 is control dependent on p1 and S2 is control dependent on p2 but not on p1.
Compiler Perspectives on Code Movement

- Two (obvious?) constraints on control dependences:
  - An instruction that is control dependent on a branch cannot be moved before the branch.
  - An instruction that is not control dependent on a branch cannot be moved to after the branch (or its execution will be controlled by the branch).
- Control dependencies relaxed to get parallelism:
  - Can occasionally move dependent loads before branch to get early start on cache miss
  - Get same effect if preserve order of exceptions (address in register checked by branch before use) and data flow (value in register depends on branch)

Where are the control dependencies?

1. Loop: LD F0,0(R1)
2. ADDD F4,F0,F2
3. SD 0(R1),F4
4. SUBI R1,R1,8
5. BEQZ R1,exit
6. LD F0,0(R1)
7. ADDD F4,F0,F2
8. SD 0(R1),F4
9. SUBI R1,R1,8
10. BEQZ R1,exit
11. LD F0,0(R1)
12. ADDD F4,F0,F2
13. SD 0(R1),F4
14. SUBI R1,R1,8
15. BEQZ R1,exit
....

When Safe to Unroll Loop?

- Example: Where are data dependencies?
  (A,B,C distinct & nonoverlapping)

```plaintext
for (i=0; i<100; i=i+1) {
  A[i+1] = A[i] + C[i]; /* S1 */
  B[i+1] = B[i] + A[i+1]; /* S2 */
}
```
1. S2 uses the value, A[i+1], computed by S1 in the same iteration.
2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
   This is a "loop-carried dependence": between iterations
- For our prior example, each iteration was distinct
  - In this case, iterations can’t be executed in parallel, Right????

Does a loop-carried dependence mean there is no parallelism???

- Consider:
  ```plaintext
  for (i=0; i< 8; i=i+1) {
    A = A + C[i]; /* S1 */
  }
  ```
  Could compute:
  ```plaintext
  "Cycle 1": temp0 = C[0] + C[1];
  temp1 = C[2] + C[3];
  temp2 = C[4] + C[5];
  temp3 = C[6] + C[7];
  "Cycle 2": temp4 = temp0 + temp1;
  temp5 = temp2 + temp3;
  "Cycle 3": A = temp4 + temp5;
  ```
  - Relies on associative nature of "+".
Trace Scheduling in VLIW

- Parallelism across IF branches vs. LOOP branches

- Two steps:
  - **Trace Selection**
    - Find likely sequence of basic blocks (trace) of (statically predicted or profile predicted) long sequence of straight-line code
  - **Trace Compaction**
    - Squeeze trace into few VLIW instructions
    - Need bookkeeping code in case prediction is wrong

- This is a form of compiler-generated speculation
  - Compiler must generate “fixup” code to handle cases in which trace is not the taken branch
  - Needs extra registers: undoes bad guess by discarding

- Subtle compiler bugs mean wrong answer vs. poorer performance; no hardware interlocks

Summary

- Interrupts and Exceptions either interrupt the current instruction or happen between instructions
  - Possibly large quantities of state must be saved before interrupting
- Machines with precise exceptions provide one single point in the program to restart execution
  - All instructions before that point have completed
  - No instructions after or including that point have completed

- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction

- Increasing length of pipe increases impact of hazards
  - Pipelining helps instruction bandwidth, not latency!

- Instruction Level Parallelism (ILP) found either by compiler or hardware.

- Loop level parallelism is easiest to see
  - SW dependencies/compiler sophistication determine if compiler can unroll loops
  - Memory dependencies hardest to determine => Memory disambiguation
  - Very sophisticated transformations available