Pipelining and Precise Interrupts
Sequential Execution Semantics

We will be studying techniques that exploit the semantics of Sequential Execution.

Sequential Execution Semantics:

instructions *appear* as if they executed in the program specified order and one after the other

Alternatively

At any given point in time we should be able to identify an instruction so that:

1. All preceding instructions have executed
2. None following has executed
Sequential Execution Semantics

• **Contract:** This is how the machine *appears to behave*
Exploiting Sequential Semantics

• The “it should appear” is the key

• The only way one can inspect execution order is via the machine’s state

This includes registers, memory and any other named storage

We will looking at techniques that relax execution order while preserving sequential execution semantics

First is PIPELINING:

  Partially Overlap Instructions

  Treat each as a sequence of micro-actions and overlap those
Dissecting Instructions

• One way or another instructions fall under the following categories:

1. **Data movement**: memory or register read and write

2. **Data manipulation**: add, subtract, etc.

3. **Control Flow**: Based on data decide what to do next, e.g., branch, jump.

How much of this and how can greatly impact the pipelinability of an instruction set architecture.
Steps of Instruction Execution

Instruction execution is not a monolithic action

There are multiple *micro-actions* involved
Pipelining: Partially Overlap Instructions

<table>
<thead>
<tr>
<th></th>
<th>Unpipelined</th>
<th>Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>time</strong></td>
<td><img src="image" alt="Unpipelined Time Diagram" /></td>
<td><img src="image" alt="Pipelined Time Diagram" /></td>
</tr>
<tr>
<td>latency</td>
<td><img src="image" alt="Unpipelined Latency Diagram" /></td>
<td><img src="image" alt="Pipelined Latency Diagram" /></td>
</tr>
<tr>
<td>1/Throughput</td>
<td><img src="image" alt="Unpipelined Throughput Diagram" /></td>
<td><img src="image" alt="Pipelined Throughput Diagram" /></td>
</tr>
<tr>
<td>instructions</td>
<td><img src="image" alt="Unpipelined Instructions Diagram" /></td>
<td><img src="image" alt="Pipelined Instructions Diagram" /></td>
</tr>
</tbody>
</table>

**Ideally:** \( T_{\text{pipeline}} = \frac{T_{\text{sequential}}}{\text{PipelineDepth}} \)

This ignores fill and drain times
Pipelining is a more general concept that can be applied to other blocks of logic or at finer than the micro-action granularities.
Sequential Semantics Preserved?

Two execution states:

1. **In-progress**: changes not visible to outsiders

2. **Committed**: changes visible
Critical Path Determines Clock Cycle

Overlap

Pick Longest Stage
Ideal Speedup

Let $T$ be the time to execute an instruction

Instruction execution requires $n$ stages, $t_1...t_n$ taking $T = \sum t_i$

W/O pipelining: $TR = \frac{1}{T} = \frac{1}{\sum t_i}$, $Latency = T = \frac{1}{TR}$

W/ n-stage pipeline: $TR = \frac{1}{\max(t_i)} \leq \frac{n}{T}$, $Latency = n \times \max(t_i) \geq T$

$$Speedup = \frac{\sum t_i}{\max(t_i)} \leq n$$

If all $t_i$ are equal, Speedup is $n$

Ideally: Want higher Performance? Use more pipeline stages
• We can just add more stages
  
  Circuits just work

• Same latency micro-actions
  
  Perfectly balanced stages

• Identical micro-actions per instruction
  
  Must perform the same steps per instruction

• Independence of micro-actions across instructions
  
  No need to wait for a previous instruction to finish
  
  No need to use the same resource at the same time
Pipelining Limits

• After a certain number of stages benefits level off and later they start diminishing

• Pipeline utility is limited by:

1. Implementation
   a. Logic Delay
   b. Clock Skew
   c. Latch Delay

2. Structures

3. Programs

2 & 3 will be called HAZARDS
Parenthetic Topic: Logic Delay FO4

• How does the speed of a gate depend on technology

• Use Fanout of 4 inverter metric

  — Measure the delay of an inverted with $C_{out}/C_{in} = 4$

• Divide speed of circuit by speed of FO4 inverter
  — Get circuit delay measured in FO4
  — Metric pretty stable, over process, temp, and voltage

Source: M. Horowitz
FO4 Inverter Delay Under Scaling

- Device performance will scale
  - FO4 delay has been linear with tech
    - Approximately 0.36 nS/μm*L_{drawn} at TT
      - (0.5nS/μm under worst-case conditions)
  - Easy to predict gate performance
    - We can measure them
      - Labs have built 0.04μm devices
    - Key issue is voltage scaling

Source: M. Horowitz
Gates Per Clock

- Clock speed has been scaling faster than base technology
- Number of FO4 delays in a cycle has been falling

- Number of gates decrease 1.4x each generation
- Caused by:
  - Faster circuit families (dynamic logic)
  - Better optimization
- Approaching a limit:
  - <16 FO4 is hard
  - <8 FO4 is very hard
Pipeline Limits: #1 Logic Delay

- \( t_L \) = logic block’s worst case delay
- \( T \) = clock period

\[ T \geq t_L \]

- Today’s Procs:
  - 6-12 2-input gates per stage
CLOCK SKEW

- Clock takes time to travel
- Arrival depends on distance/load
- Skew amongst different connections
- Creates additional constraints

Clock Skew
Clock Skew contd.

- Output from stage with late clock feeds stage with early clock

Worst case scenario:
Pipelining Limits: #3 Latch Overhead

- Latch takes time
  - Setup Time
    data must stay stable before clock edge
  - Hold Time
    data must stay stable after clock edge

\[ T \]

\[ t_l \]

computation starts

should complete before this

setup  hold
Impact of Clock Skew and Latch Overheads

let $X$ be extra delay per stage for

- latch overhead
- clock/data skew

$X$ limits the useful pipeline depth

With n-stage pipeline (all $t_i$ equal) ($T = n \times t$)

- throughput $= \frac{1}{X + t} < \frac{n}{T}$
- latency $= n \times (X + t) = n \times X + T$
- speedup $= \frac{T}{(X + t)} \leq n$

Real pipelines usually do not achieve this due to Hazards
Ideal speedup with per stage overheads

\[ T = 500. \text{ A: } X=100, \text{ B: } X=10 \]
Cost/Performance Tradeoff

Cost = Stages x Latch_Cost + Base Cost

Performance = 1 / (Overhead_{per\_stage} + Cycle / Stages)

Source: P. Kogge: The architecture of pipelined computers.

Cost / Performance = LatchCost x CycleTime +

BaseCost x Overhead +

LatchCost x Overhead x Stages +

BaseCost x CycleTime / Stages

Stages_{Optimal} = \sqrt{\frac{\text{BaseCost} \times \text{CycleTime}}{\text{LatchCost} \times \text{Overhead}}}
Optimal Number of Stages

\( n = \text{stages}, \text{ source C. Kozyrakis} \)
Cost/Performance trade-off

Cycle = 500, BaseCost = 200

A: Overhead = 10, LatchCost = 20 / B: Overhead = 20 & LatchC = 40

As latchcost and overhead become larger w/ respect to base cost, the number of stages must be reduced -- Deep pipelines no good.
Pipeline Idealism

• We can just add more stages
  Circuits just work

• Same latency micro-actions
  Perfectly balanced stages

• Identical micro-actions per instruction
  Must perform the same steps per instruction

• Independence of micro-actions across instructions
  No need to wait for a previous instruction to finish
  No need to use the same resource at the same time
Skew can be your friend

time borrowing

revisited

i IN

i Out/i+1 IN

i+1 Out
Simple pipelines

F- fetch, D - decode, X - execute, M - memory, W - writeback

Classic 5-stage Pipeline
MIPS micro-actions per instruction

- **integer/logic operations**
  
  `add $1, $2, $3` --> read 2 regs, write one

- **branches**
  
  `beq $1, $2, LALA` --> read 2 regs, compare, change PC

- **load/stores**
  
  `lw $1, 10($3)` --> read 1 reg, add, read memory, write reg

  `sw $1, 10($3)` --> read 2 regs, add, write memory

- **special ops: syscall, jumps, call/return**
  
  read at most 1 reg, write at most 1 reg, change PC
Non-Pipelined Implementation
Pipelined Implementation

Pipelined Implementation: Ideally 5x performance
Hazards

- conditions that lead to incorrect behavior if not fixed

Structural Hazard
- two different instructions use same resource in same cycle

Data Hazard
- two different instructions use same storage
- must appear as if the instructions execute in correct order

Control Hazard
- one instruction affects which instruction is next
Pipelining as Datapaths in Time

Useful for identifying hazards
Structural Hazards

When two or more different instructions want to use the same hardware resource in the same cycle

- e.g., load and stores use the same memory port as IF
Dealing with Structural Hazards

Stall:

+ low cost, simple
– decrease IPC
• use for rare case

Pipeline Hardware Resource:

• useful for multicycle resources
+ good performance
– sometimes complex e.g., RAM
– Example 2-stage cache pipeline: decode, read or write data (wave pipelining - generalization)
Dealing with Structural Hazards

Replicate resource

+ good performance
– increases cost
– increased interconnect delay ?

• use for cheap or divisible resources
Impact of ISA on Structural Hazards

Structural hazards are reduced

- If each instruction uses a resource at most once
- Always in same pipeline stage
- For one cycle

Many RISC ISAs designed with this in mind

RISC = Reduced Instruction Set Architecture

Today: Load/Store Architectures, not necessarily few instructions
Data Hazards

When two different instructions use the same storage location, it must appear as if they executed in sequential order.

```
add r1, r2, --
sub r2, --, r1
add r3, r1, --
or r1, --, --
```

- **read-after-write** (RAW, true dependence) -- real
- **write-after-read** (WAR, anti-dependence) -- artificial
- **write-after-write** (WAW, output-dependence) -- artificial
- **read-after-read** (no hazard)
Dependences in a 5-Stage Pipeline

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of register $2$:</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
</tbody>
</table>

Program execution order (in instructions):
- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)
Examples of RAW

- `add r1, --, --`  
  IF ID EX MEM WB  
  `r1 written`

- `sub --, r1, --`  
  IF ID EX MEM WB  
  `r1 read - not OK`

- `load r1, --, --`  
  IF ID EX MEM WB  
  `r1 written`

- `sub --, r1, --`  
  IF ID EX MEM WB  
  `r1 read - not OK`

- `sw r1, 100(r2)`  
  IF ID EX MEM WB

- `lw r1, 100(r2)`  
  IF ID EX MEM WB  
  `OK`

unless 100(r2) is the PC of the load (self-modifying code)
Simple Solution to RAW

Hardware detects RAW and stalls

\[
\begin{align*}
\text{add } r1, --, -- & \quad \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
\text{sub } --, r1, -- & \quad \text{IF} \quad \text{stall} \quad \text{stall} \quad \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB}
\end{align*}
\]

+ low cost, simple

– reduces IPC

Maybe we should try to minimize stalls
Stall Methods

Compare ahead in pipe

- if rs1(EX) == Rd(MEM) || rs2(EX) == Rd(MEM) then stall
- assumes MEM instr is a load, EX instr is ALU

Use register reservation bits:

one bit per register

set at ID stage  

check source Reg bit  

stall if reserved

clear at MEM stage
Minimizing RAW stalls

Bypass or Forward or Short-Circuit

\[ \text{data available} \quad \text{r1 written} \]

\[
\begin{array}{cc}
\text{add r1, --, -- } & \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
\text{sub --, r1, --} & \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
\end{array}
\]

Use data before it is in register

+ reduces/avoids stalls

– complex

– Deeper pipelines -> more places to bypass from

• crucial for common RAW hazards
Bypass

Interlock logic

- detect hazard
- bypass correct result to ALU

Hardware detection requires extra hardware

- instruction latches for each stage
- comparators to detect the hazards
Bypass Example
Mux control

- if \( \text{insn(EX)} \) uses immediate then select IMM
- else if \( \text{rs2(EX)} == \text{rd(MEM)} \) then \( \text{ALUOUT(MEM)} \)
- else if \( \text{rs2(EX)} == \text{rd(WB)} \) then \( \text{ALUOUT(WB)} \)
- else select B
Stall Method w/ Bypass paths

- Use register reservation bits:
  - is the result available the next cycle
  - Only loads set the bits --> instructions that cannot be bypassed

- Stall Method w/ Bypass paths

  one bit per register

  set at ID stage           clear at EX stage

  loads reserve at ID stage  check source Reg bit
  release at EX stage       stall if reserved
RAW solutions

Hybrid (i.e., stall and bypass) solutions required sometimes

load r1, --, --       IF   ID   EX   MEM   WB
sub --, r1, --         stall IF   ID   EX   MEM   WB

DLX has one cycle bubble if load result used in next instruction

Try to separate stall logic from bypass logic

• avoid irregular bypasses
Pipeline Scheduling - Compilers can Help

Instructions scheduled by compiler to reduce stalls

```plaintext
a = b + c;  d = e + f -- Prior to scheduling

lw Rb, b
lw Rc, c
stall
add Ra, Rb, Rc
sw a, ra
lw Re, e
lw Rf, f
stall
sub Rd, Re, Rf
sw d, Rd
```
Pipeline Scheduling

After scheduling

\[
\begin{align*}
\text{lw} & \quad \text{Rb}, \quad b \\
\text{lw} & \quad \text{Rc}, \quad c \\
\text{lw} & \quad \text{Re}, \quad e \\
\text{add} & \quad \text{Ra}, \quad \text{Rb}, \quad \text{Rc} \\
\text{lw} & \quad \text{Rf}, \quad f \\
\text{sw} & \quad a, \quad \text{ra} \\
\text{sub} & \quad \text{Rd}, \quad \text{Re}, \quad \text{Rf} \\
\text{sw} & \quad d, \quad \text{Rd1} \\
\end{align*}
\]

No Stalls
Delayed Load

Avoid hardware solutions - Let the compiler deal with it

Instruction Immediately after load can’t/shouldn’t see load result

Compiler has to fill in the delay slot - NOP might be necessary

\[
\begin{align*}
\text{UNSCHEDULED} & \quad \text{SCHEDULED} \\
\text{lw Rb, b} & \quad \text{lw Rb, b} \\
\text{lw Rc, c} & \quad \text{lw Rc, c} \\
\text{nop} & \quad \text{nop} \\
\text{add Ra, Rb, Rc} & \quad \text{add Ra, Rb, Rc} \\
\text{sw a, Ra} & \quad \text{lw Rf, f} \\
\text{lw Re, efs} & \quad \text{sw a, Ra} \\
\text{lw Rf, f} & \quad \text{lw Rf, f} \\
\text{nop} & \quad \text{sub Rd, Re, Rf} \\
\text{add Rd, Re, Rf \ldots} & \quad \text{sw d, Rd}
\end{align*}
\]
Other Data Hazards: WAR

WAR

add r1, r2, --
sub r2, --, r1
or r1, --, --

Not possible in MIPS - read early write late

Consider late read then early write: delayed reg reads for stores
ALU ops writeback at EX stage
MEM takes two cycles and stores need source reg after 1 cycle

sw r1, --   IF  ID  EX  MEM1  MEM2  WB
add r1, --, --   IF  ID  EX  MEM1  MEM2  WB
also:    MUL --, 0(r2), r1
lw --, (r1++)
Other Data Hazards: WAW

WAW

Not in MIPS: register writes are in order
consider slow then fast operation

update r1

update r1 not OK
# Control Hazards

When an instruction affects which instruction execute next or changes the PC

- `sw $4, 0($5)`
- `bne $2, $3, loop`
- `sub -, -, -`

<table>
<thead>
<tr>
<th></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><strong>sw</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>bne</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>??</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control Hazards

Handling control hazards is very important

VAX e.g.,
- Emer and Clark report 39% of instr. change the PC
- Naive solution adds approx. 5 cycles every time
- *Or, adds 2 to CPI or ~20% increase*

DLX e.g.,
- H&P report 13% branches
- Naive solution adds 3 cycles per branch
- *Or, 0.39 added to CPI or ~30% increase*
- **RULE of THUMB: 1/5 instructions is a BRANCH**
Handling Control Hazards

Move control point earlier in the pipeline

- Find out whether branch is taken earlier
- Compute target address fast

Both need to be done

e.g., in ID stage

- target := PC + immediate
- if (Rs1 op 0) PC := target
### Handling Control Hazards

<table>
<thead>
<tr>
<th></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>N: sw</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N+1: bne</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N+2: add</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y: sub</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Implies only one cycle bubble but**

- special PC adder required
ISA and Control Hazard

Comparisons in ID stage

- must be fast
- can’t afford to subtract
- compares with 0 are simple
- gt, lt test sign-bit
- eq, ne must OR all bits

More general conditions need ALU

- MIPS uses conditional sets
Handling Control Hazards

Branch prediction

- guess the direction of branch
- minimize penalty when right
- may increase penalty when wrong

Techniques

- static - by compiler
- dynamic - by hardware
- MORE ON THIS LATER ON
Handling Control Hazards

Static techniques

- predict always not-taken
- predict always taken
- predict backward taken
- predict specific opcodes taken
- delayed branches

Dynamic techniques

- Discussed with ILP
Handling Control Hazards

Predict not-taken always

this is the interesting column

<table>
<thead>
<tr>
<th></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>i</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if taken then squash (aka abort or rollback)

- will work only if no state change until branch is resolved
- Simple 5-stage Pipeline, e.g., DLX - ok - why?
- Other pipelines, e.g., VAX - autoincrement addressing?
Handling Control Hazards

Predict taken always

For DLX must know target before branch is decoded
- can use prediction
- special hardware for fast decode

Execute both paths - hardware/memory b/w expensive
Handling Control Hazards

Delayed branch - execute next instruction whether taken or not

- i: beqz r1, #8
- i+1: sub --, --, --
- . . . . .
- i+8: or --, --, -- (reused by RISC invented by microcode)
Filling in Delay slots

Fill with an instr before branch
  • When? if branch and instr are independent.
  • Helps? always

Fill from target (taken path)
  • When? if safe to execute target, may have to duplicate code
  • Helps? on taken branch, may increase code size

Fill from fall-through (not-taken path)
  • when? if safe to execute instruction
  • helps? when not-taken
Filling in Delay Slots cont.

From Control-Independent code:
that’s code that will be *eventually* visited no matter where the branch goes

Nullifying or Cancelling or Likely Branches:
Specify when delay slot is execute and when is squashed

*Why?* Increase fill opportunities

**Major Concern w/ DS:** Exposes implementation optimization
Comparison of Branch Schemes

Cond. Branch statistics - DLX H&P

- 14%-17% of all insts (integer)
- 3%-12% of all insts (floating-point)
- Overall 20% (int) and 10% (fp) control-flow insts.
- About 67% are taken

Branch-Penalty = \%branches \times

\((\%taken \times \text{taken-penalty} + \%\text{not-taken} \times \text{not-taken-penalty})\)
## Comparison of Branch Schemes

<table>
<thead>
<tr>
<th>scheme</th>
<th>taken penalty</th>
<th>not-taken pen.</th>
<th>CPI penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>3</td>
<td>3</td>
<td>0.420</td>
</tr>
<tr>
<td>fast branch</td>
<td>1</td>
<td>1</td>
<td>0.140</td>
</tr>
<tr>
<td>not-taken</td>
<td>1</td>
<td>0</td>
<td>0.091</td>
</tr>
<tr>
<td>taken</td>
<td>0</td>
<td>1</td>
<td>0.049</td>
</tr>
<tr>
<td>delayed branch</td>
<td>0.5</td>
<td>0.5</td>
<td>0.070</td>
</tr>
</tbody>
</table>

Assuming: branch\% = 14\%, taken\% = 65\%, 50\% delay slots are filled w/ useful work

ideal CPI is 1
Impact of Pipeline Depth

Assume that now penalties are doubled

For example we double clock frequency

<table>
<thead>
<tr>
<th>scheme</th>
<th>taken penalty</th>
<th>not-taken pen.</th>
<th>CPI penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>6</td>
<td>6</td>
<td>0.840</td>
</tr>
<tr>
<td>fast branch</td>
<td>2</td>
<td>2</td>
<td>0.280</td>
</tr>
<tr>
<td>not-taken</td>
<td>2</td>
<td>0</td>
<td>0.182</td>
</tr>
<tr>
<td>taken</td>
<td>0</td>
<td>2</td>
<td>0.098</td>
</tr>
<tr>
<td>delayed branch</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Precise Interrupts
Interrupts

Examples:

- power failing, arithmetic overflow
- I/O device request, OS call, page fault
- Invalid opcode, breakpoint, protection violation

Interrupts (aka faults, exceptions, traps) often require

- surprise jump (to vectored address)
- linking return address
- saving of PSW (including CCs)
- state change (e.g., to kernel mode)
Classifying Interrupts

1a. synchronous
   • function of program state (e.g., overflow, page fault)

1b. asynchronous
   • external device or hardware malfunction

2a. user request
   • OS call

2b. coerced
   • from OS or hardware (page fault, protection violation)
Classifying Interrupts

3a. User Maskable
   User can disable processing

3b. Non-Maskable
   User cannot disable processing

4a. Between Instructions
   Usually asynchronous

4b. Within an instruction
   Usually synchronous - Harder to deal with

5a. Resume
   As if nothing happened? Program will continue execution

5b. Termination
Restartable Pipelines

• Interrupts within an instruction are not catastrophic
• Most machines today support this
  Needed for virtual memory
• Some machines did not support this
  Why?
    Cost
    Slowdown

Key: Precise Interrupts
Will return to this soon
First let’s consider a simple 5-stage pipeline
Handling Interrupts

Precise interrupts (sequential semantics)

- Complete instructions before the offending instr
- Squash (effects of) instructions after
- Save PC (& next PC with delayed branches)
- Force trap instruction into IF

Must handle simultaneous interrupts

- IF, M - memory access (page fault, misaligned, protection)
- ID - illegal/privileged instruction
- EX - arithmetic exception
Interrupts: Data Memory Page Fault

E.g., data page fault

<table>
<thead>
<tr>
<th></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>i</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>&lt;- page fault</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>F</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>trap -&gt;</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x+1</td>
<td>trap handler -&gt;</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Interrupts

Preceding instructions already complete

Squash succeeding instructions
  • prevent them from modifying state (registers, CC, memory)

trap instruction jumps to trap handler

hardware saves PC in IAR

trap handler must save IAR
Interrupts: Arithmetic Exception

E.g., arithmetic exception

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
i & F & D & X & M & W \\
i+1 & F & D & X & M & W \\
i+2 & F & D & X & \text{exception} \\
i+3 & F & D & \text{squash} \\
i+4 & F & \text{squash} \\
x & \text{trap} \rightarrow & F & D & X & M & W \\
x+1 & \text{trap handler} \rightarrow & F & D & X & M & W \\
\end{array}
\]
Interrupts: Instruction Page Fault

E.g., Instruction fetch page fault

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
i & F & D & X & M & W \\
i+1 & F & D & X & M & W \\
i+2 & F & D & X & M & W \\
i+3 & F & D & X & M & W \\
i+4 & F & D & X & M & W \\
x & trap -> & F & D & X & M & W \\
x+1 & trap handler -> & F & D & X & M & W \\
\end{array}
\]
Interrupts

Let preceding instructions complete

No succeeding instructions

What happens if i+3 causes a data page fault?
Out-of-Order Interrupts

Out-of-order interrupts

• which page fault should we take?

```
   1  2  3  4  5  6  7  8  9
i  F  D  X  M  W
i+1 F  D  X  M  W
i+2 F  D  X  M  W
i+3 F  D  X  M  W
```

page fault (Mem)

page fault (fetch)
Out-of-Order Interrupts

Post interrupts: **In-order handling**

- check interrupt bit on entering WB
- precise interrupts
- longer latency

Handle immediately: **out-of-order handling**

- not fully precise
- interrupt may occur in order different from sequential CPU
- may cause implementation headaches
- Violates Sequential Model
- But offers high performance.
Interrupts

Other complications

- odd bits of state (e.g., CC)
- early-writes (e.g., autoincrement)
- instruction buffers and prefetch logic
- dynamic scheduling
- out-of-order execution

Interrupts come at random times

Both Performance and Correctness

- frequent case must perform well
- rare case MUST work correctly
Delayed Branches and Interrupts

What happens on interrupt while in delay slot

- next instruction is not sequential

Solution #1: save multiple PCs

- save current and next PC
- special return sequence, more complex hardware

Solution #2: single PC plus

- branch delay bit
- PC points to branch instruction
- SW Restrictions
Multicycle operations

Not all operations complete in 1 cycle

- FP slower than integer
- 2-4 cycles multiply or add
- 20-50 cycles divide

Extend DLX pipeline

- EX stage repeated multiple times
- multiple, parallel functional units
  - not pipelined for now
Handling Multicycle Operations

Four functional units

- EX: integer
- E*: FP/integer multiplier
- E+: FP adder
- E/: FP/integer divider

Assume

- EX takes 1 cycle and all FP take 4 cycles
- separate integer and FP registers
- all FP arithmetic in FP registers

Worry about hazards

- structural, RAW (forwarding), WAR/WAW (between I & FP)
## Simple Multicycle Example

<table>
<thead>
<tr>
<th></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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fp*</td>
<td>F</td>
<td>D</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fp/</td>
<td>F</td>
<td>D</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fp/</td>
<td>F</td>
<td>D</td>
<td>--</td>
<td>--</td>
<td>E/</td>
<td>E/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fp*</td>
<td>F</td>
<td>--</td>
<td>--</td>
<td>D</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) **Interrupts? no WAW**

(2) **no WB conflict**

(3) **structural**

(4) **in-order issue**
Simple Multicycle Example

Notes:

• (1) - no WAW but complicates interrupts
• (2) - no WB conflict
• (3) - stall forced by structural hazard
• (4) - stall forced by in-order issue

Different FP operation times are possible
• Makes FP WAW hazards possible
• Further complicates interrupts
FP Instruction Issue

Check for structural hazards

- wait until functional unit is free

Check for RAW - wait until

- source regs are not used as destinations by instrs in EX_i

Check for forwarding

- bypass data from MEM or WB if needed

What about overlapping instructions?

- contention in WB
- possible WAR/WAW hazards
- interrupt headaches
Overlapping Instructions

Contention in WB

- static priority
- e.g., FU with longest latency
- instructions stall after issue

WAR hazards

- always read registers at same pipe stage

WAW hazards

- divf f0, f2, f4 followed by subf f0, f8, f10
- stall subf or abort divf’s WB
Multicycle Operations

Problems with interrupts

- DIVF f0, f2, f4
- ADDF f2, f8, f10
- SUBF f6, f4, f10

ADDF completes before DIVF

- Out-Of-Order completion
- Possible imprecise interrupts

What if divf excepts after addf/subf complete?

Precice Interrupts Paper
In-Order Completion

Time as an infinite sequence of WB slots:

As instructions come in (decode) they reserve all time slots up to their completion time: e.g., an add, followed by a sub in a 5-stage pipeline:

Time --> ADD reserves 3 slots, done at decode

Time --> SUB reserves 3 (two already reserved plus one more)
In-Order Completion: Long Latency Instr

mul f0, f1, 10
  • add r1, r2, r3

Time --> MULF reserves 6 slots, at time 0

Time --> ADD tries to reserve 3 slots, at time 1, slot 1+3 = 4 is taken, so is 5, and 6
Implementation: Result Shift Register

• Simple solution: Modify state only when all preceding insts. are known to be exception free.

Mechanism: Result Shift Register

<table>
<thead>
<tr>
<th>stage</th>
<th>FU</th>
<th>DR</th>
<th>V</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>div</td>
<td>R1</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>add</td>
<td>R2</td>
<td>1</td>
<td>1001</td>
</tr>
</tbody>
</table>

Reserve all stages for the duration of the instruction

Think of this as a wheel over the infinite sequence of WB slots
Out-of-Order Completion

• Simple approach: Performance loss

• High-Performance approach:
  — Allow out-of-order WBs
  — Upon an interrupt cleanup and then present state to the user

interrupt detected

Sequentially consistent state
Handler executes
Cleanup Approaches: History File

keep a log of all changes made out-of-order

e.g., I updated register R10 and before I did it’s value was 20

To cleanup, revert all changes, do so in reverse program order*

this will be needed for out-of-order execution w/ register renaming

divf f1, __, __ save previous value of f1
add r1, __, __ save previous value of r1
sub r2, r1, __ save previous value of r2

\[
\begin{array}{cccccccc}
\text{divf f1,} & \text{F} & \text{D} & \text{X} & \text{X} & \text{X} & \text{X} & \text{X} & \text{W} \\
\text{add r1,} & \text{F} & \text{D} & \text{X} & \text{M} & \text{W} & \text{C} \\
\text{sub r2,} & \text{F} & \text{D} & \text{X} & \text{M} & \text{W} & \text{C} \\
\end{array}
\]
History File Contd.

Cleaning up after an exception

```
divf f1,
add r1,
sub r2,
```
History Buffer

- Allow out-of-order register file updates
- At decode record current value of target register in reorder buffer entry.
- On commit: do nothing
- On exception: scan following reorder buffer entries restoring register values

![Diagram of RF and HB with exception and dst. connections]

Moshovos  ECE1773  97
Future File

- Keep two states:
  - Architectural: Updated in-order always consistent sequentially
  - Future: Updated out-of-order can be inconsistent
- On an exception reset the Future and use the Architectural
- For regular execution use Future and if empty, Architectural
Instructions do TWO writes
### Future File Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>F</th>
<th>D</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>divf f1,</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>/F</td>
</tr>
<tr>
<td>add r1,</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>F</td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>sub r2,</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>F</td>
<td></td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
Reorder Buffer: Keeping Order

- Out-of-order completion
- Commit: Write results to register file or memory
- Reorder buffer holds not yet committed state

result shift reg.

<table>
<thead>
<tr>
<th>st.</th>
<th>FU</th>
<th>V</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>div</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

reorder buffer

<table>
<thead>
<tr>
<th>tag</th>
<th>DR</th>
<th>Result</th>
<th>V</th>
<th>E</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

div (4 cycles)
add (1 cycle)
Future File

• Two register files:
  One updated out-of-order (FUTURE)
    
    assume no exceptions will occur
  One updated in order (ARCHITECTURAL)
• Advantage: No delay to restore state on exception