Overview

Wrap-Up of PIPE Design

- Exceptional conditions
- Performance analysis
- Fetch stage design

Modern High-Performance Processors

- Out-of-order execution
Exceptions

- Conditions under which processor cannot continue normal operation

Causes

- Halt instruction (Current)
- Bad address for instruction or data (Previous)
- Invalid instruction (Previous)

Typical Desired Action

- Complete some instructions
  - Either current or previous (depends on exception type)
- Discard others
- Call exception handler
  - Like an unexpected procedure call

Our Implementation

- Halt when instruction causes exception
Exception Examples

Detect in Fetch Stage

jmp $-1               # Invalid jump target
.byte 0xFF            # Invalid instruction code
halt                  # Halt instruction

Detect in Memory Stage

irmovl $100,%eax
rmmovl %eax,0x10000(%eax)  # invalid address
Exceptions in Pipeline Processor #1

```plaintext
# demo-excl.ys
irmovl $100,%eax
rmmovl %eax,0x10000(%eax)  # Invalid address
nop
.byte 0xFF                  # Invalid instruction code
```

0x000: `irmovl $100,%eax`
0x006: `rmmovl %eax,0x10000(%eax)`  # Invalid address
0x00c: `nop`
0x00d: `.byte 0xFF`

**Desired Behavior**

- `rmmovl` should cause exception
- Following instructions should have no effect on processor state
Exceptions in Pipeline Processor #2

`# demo-exc2.ys`

```assembly
0x000:  xorl %eax,%eax     # Set condition codes
0x002:  jne t              # Not taken
0x007:  irmovl $1,%eax
0x00d:  irmovl $2,%edx
0x013:  halt
0x014:  t: .byte 0xFF      # Target
```

### Desired Behavior

- No exception should occur
Maintaining Exception Ordering

- Add status field to pipeline registers
- Fetch stage sets to either “AOK,” “ADR” (when bad fetch address), “HLT” (halt instruction) or “INS” (illegal instruction)
- Decode & execute pass values through
- Memory either passes through or sets to “ADR”
- Exception triggered only when instruction hits write back
Exception Handling Logic

Fetch Stage

# Determine status code for fetched instruction
int f_stat = [
    imem_error: SADR;
    !instr_valid : SINS;
    f_icode == IHALT : SHLT;
    1 : SAOK;
];

Memory Stage

# Update the status
int m_stat = [
    dmem_error : SADR;
    1 : M_stat;
];

Writeback Stage

int Stat = [
    # SBUB in earlier stages indicates bubble
    W_stat == SBUB : SAOK;
    1 : W_stat;
];
Side Effects in Pipeline Processor

# demo-exc3.ys
irmovl $100,%eax
rmmovl %eax,0x10000(%eax)  # invalid address
addl %eax,%eax  # Sets condition codes

 Desired Behavior

- rmmovl should cause exception
- No following instruction should have any effect
Avoiding Side Effects

Presence of Exception Should Disable State Update

- Invalid instructions are converted to pipeline bubbles
  - Except have stat indicating exception status
- Data memory will not write to invalid address
- Prevent invalid update of condition codes
  - Detect exception in memory stage
  - Disable condition code setting in execute
  - Must happen in same clock cycle
- Handling exception in final stages
  - When detect exception in memory stage
    » Start injecting bubbles into memory stage on next cycle
  - When detect exception in write-back stage
    » Stall excepting instruction
- Included in HCL code
Control Logic for State Changes

Setting Condition Codes

# Should the condition codes be updated?
bool set_cc = E_icode == IOPL &&
    # State changes only during normal operation
    !m_stat in { SADR, SINS, SHLT }
    && !W_stat in { SADR, SINS, SHLT };

Stage Control

- Also controls updating of memory

# Start injecting bubbles as soon as exception passes through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT }
    || W_stat in { SADR, SINS, SHLT };

# Stall pipeline register W when exception encountered
bool W_stall = W_stat in { SADR, SINS, SHLT };

Rest of Real-Life Exception Handling

Call Exception Handler

- Push PC onto stack
  - Either PC of faulting instruction or of next instruction
  - Usually pass through pipeline along with exception status
- Jump to handler address
  - Usually fixed address
  - Defined as part of ISA

Implementation

- Haven’t tried it yet!
Performance Metrics

Clock rate

- Measured in Gigahertz
- Function of stage partitioning and circuit design
  - Keep amount of work per stage small

Rate at which instructions executed

- CPI: cycles per instruction
- On average, how many clock cycles does each instruction require?
- Function of pipeline design and benchmark programs
  - E.g., how frequently are branches mispredicted?
CPI for PIPE

CPI ≈ 1.0

- Fetch instruction each clock cycle
- Effectively process new instruction almost every cycle
  - Although each individual instruction has latency of 5 cycles

CPI > 1.0

- Sometimes must stall or cancel branches

Computing CPI

- C clock cycles
- I instructions executed to completion
- B bubbles injected (C = I + B)

\[
\text{CPI} = \frac{C}{I} = \frac{(I+B)}{I} = 1.0 + \frac{B}{I}
\]

- Factor B/I represents average penalty due to bubbles
CPI for PIPE (Cont.)

\[ B/I = LP + MP + RP \]

- **LP**: Penalty due to load/use hazard stalling
  - Fraction of instructions that are loads: 0.25
  - Fraction of load instructions requiring stall: 0.20
  - Number of bubbles injected each time: 1
  \[ \Rightarrow LP = 0.25 \times 0.20 \times 1 = 0.05 \]

- **MP**: Penalty due to mispredicted branches
  - Fraction of instructions that are cond. jumps: 0.20
  - Fraction of cond. jumps mispredicted: 0.40
  - Number of bubbles injected each time: 2
  \[ \Rightarrow MP = 0.20 \times 0.40 \times 2 = 0.16 \]

- **RP**: Penalty due to return instructions
  - Fraction of instructions that are returns: 0.02
  - Number of bubbles injected each time: 3
  \[ \Rightarrow RP = 0.02 \times 3 = 0.06 \]

Net effect of penalties: 0.05 + 0.16 + 0.06 = 0.27
\[ \Rightarrow CPI = 1.27 \quad \text{(Not bad!)} \]
Fetch Logic Revisited

During Fetch Cycle

1. Select PC
2. Read bytes from instruction memory
3. Examine icode to determine instruction length
4. Increment PC

Timing
- Steps 2 & 4 require significant amount of time
Standard Fetch Timing

- Must Perform Everything in Sequence
- Can’t compute incremented PC until know how much to increment it by
A Fast PC Increment Circuit

- High-order 29 bits
- Low-order 3 bits
- MUX
  - 0
  - 1
- 29-bit incrementer
- 3-bit adder
- incrPC
- carry
- need_regids
- need_ValC
- PC
- Fast
- Slow
Modified Fetch Timing

29-Bit Incrementer

- Acts as soon as PC selected
- Output not needed until final MUX
- Works in parallel with memory read

Select PC

Mem. Read

need_regids, need_valC

3-bit add

Incrementer

MUX

1 clock cycle

Standard cycle
More Realistic Fetch Logic

**Fetch Box**

- Integrated into instruction cache
- Fetches entire cache block (16 or 32 bytes)
- Selects current instruction from current block
- Works ahead to fetch next block
  - As reaches end of current block
  - At branch target
Modern CPU Design

Instruction Control

Fetch Control

Instruction Decode

Instruction Cache

Retirement Unit

Register File

Register Updates

Prediction OK?

Address

Instructions

Operations

Functional Units

Integer/Branch

General Integer

FP Add

FP Mul/Div

Load

Store

Data Cache

Operation Results

Addr.

Data

Data

Data

Addr.
Instruction Control

Grabs Instruction Bytes From Memory
- Based on Current PC + Predicted Targets for Predicted Branches
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

Translates Instructions Into Operations
- Primitive steps required to perform instruction
- Typical instruction requires 1–3 operations

Converts Register References Into Tags
- Abstract identifier linking destination of one operation with sources of later operations
Multiple functional units
- Each can operate independently

Operations performed as soon as operands available
- Not necessarily in program order
- Within limits of functional units

Control logic
- Ensures behavior equivalent to sequential program execution
**CPU Capabilities of Intel iCore7**

**Multiple Instructions Can Execute in Parallel**
- 1 load
- 1 store
- 1 FP multiplication or division
- 1 FP addition
- > 1 integer operation

**Some Instructions Take > 1 Cycle, but Can be Pipelined**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer Divide</td>
<td>11—21</td>
<td>5—13</td>
</tr>
<tr>
<td>Double/Single FP Multiply</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Double/Single FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Double/Single FP Divide</td>
<td>10—15</td>
<td>6—11</td>
</tr>
</tbody>
</table>
iCore Operation

Translates instructions dynamically into “Uops”
- ~118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine
- Uop executed when
  - Operands available
  - Functional unit available
- Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources
Critical to Performance

- Typically 11–15 cycle penalty for misprediction

Branch Target Buffer

- 512 entries
- 4 bits of history
- Adaptive algorithm
  - Can recognize repeated patterns, e.g., alternating taken–not taken

Handling BTB misses

- Detect in ~cycle 6
- Predict taken for negative offset, not taken for positive
  - Loops vs. conditionals
Example Branch Prediction

Branch History

- Encode information about prior history of branch instructions
- Predict whether or not branch will be taken

State Machine

- Each time branch taken, transition to right
- When not taken, transition to left
- Predict branch taken when in state Yes! or Yes?
Processor Summary

Design Technique

- Create uniform framework for all instructions
  - Want to share hardware among instructions
- Connect standard logic blocks with bits of control logic

Operation

- State held in memories and clocked registers
- Computation done by combinational logic
- Clocking of registers/memories sufficient to control overall behavior

Enhancing Performance

- Pipelining increases throughput and improves resource utilization
- Must make sure to maintain ISA behavior