The Processor – Exceptions, Parallelism, Fallacies

Alexander Nelson
March 29, 2021

University of Arkansas - Department of Computer Science and Computer Engineering
Full datapath with pipelining/forwarding/hazard detection detection
Exceptions
To make a computer with automatic program-interruption facilities behave [sequentially] was not an easy matter, because the number of instructions in various stages of processing when an interrupt signal occurs may be large.

–Fred Brooks, Jr.
Exception/Interrupt – An unscheduled event that disrupts program execution (e.g. Overflow detection)
Interrupt – An exception that arises outside of the flow of the processor

What is the difference?
My take – All exceptions are interrupts, but not all interrupts are exceptions
Exceptions

The book’s take (MIPS Convention):

<table>
<thead>
<tr>
<th>Type of event</th>
<th>From where?</th>
<th>MIPS terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device request</td>
<td>External</td>
<td>Interrupt</td>
</tr>
<tr>
<td>Invoke the operating system from user program</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Arithmetic overflow</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Using an undefined instruction</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Hardware malfunctions</td>
<td>Either</td>
<td>Exception or interrupt</td>
</tr>
</tbody>
</table>

Detecting exception conditions is critical to correct performance
Designing exception flow after the fact reduces performance
Instead, consider exceptions in the design phase
Exceptions managed by a System Control Coprocessor (CP0)

What is its purpose?
Stop current instruction(s), transfer control

Process:

- Save PC of interrupted instruction
- Save indication of the problem
- Jump to handler code
Handling Exceptions in MIPS

Exceptions managed by a System Control Coprocessor (CP0)

What is its purpose?
Stop current instruction(s), transfer control

Process:

- Save PC of interrupted instruction
  - In MIPS – Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS – “Cause” register
- Jump to handler code
Exceptions in Our Design

Let’s consider our current designed machine
What exceptions can occur?

- Undefined Opcode
- Arithmetic Overflow

Let’s give them a “cause” register:
Two potential causes = 1 bit
0 = undefined opcode, 1 = overflow

Let’s give them vector addresses:
Undefined opcode = 0x8000 0000
Arithmetic overflow = 0x8000 0180
What do we need to change?
Exception Handler

Procedure:

- Read cause, transfer to relevant handler
- Determine actions required
- If exception handled and restartable
  - Take Corrective action
  - Use EPC to return to program
- Else:
  - Terminate program
  - Report error using EPC, cause, etc.
How to handle exceptions in a pipelined implementation?
Treat exception as another control hazard!

**Review:** Control Hazard = Need a decision to finish before executing new instructions

Consider overflow on add in EX stage:  
\textit{add} \, $1, \, $2, \, $1

- Prevent $1 from being overwritten (clobbered)
- Complete previous instructions (through WB)
- Flush add & subsequent instructions
- Set Cause & EPC reg. values
- Transfer control to handler
Restartable Exceptions:

- Pipeline can flush the instruction
- Handler executes, then returns to the instruction
  - Refetched and executed from scratch

PC + 4 saved in EPC register

- Handler must identify the “causing” instruction
Example Exception

Consider the following instructions

//Exception on add (4C)
40 sub $11, $2, $4
44 and $12, $2, $5
48 or $13, $2, $6
4C add $1, $2, $1
50 slt $15, $6, $7
54 lw $16, 50($7)
...

//Handler
800000180 sw $25, 1000($0)
800000184 sw $26, 1004($0)
Example Exception

lw $16, 50($7)

slt $15, $6, $7

add $1, $2, $1

or $13, . . . and $12, . . .
Flush "add" & subsequent instructions, store cause & EPC
Transfer control to handler
So far, we have only considered 2 exception types
5 were defined in earlier table
There are more exceptions that you may encounter

We have a 5-stage pipeline = 5 instructions operating concurrently
What if multiple exceptions occur at the same time?
What if multiple exceptions occur at the same time?

Simple approach – Deal with earliest exception first!

- Flush subsequent instructions

In complex pipelines:

- Multiple instructions may be issued per cycle
- Allowed for out-of-order completion
- Maintaining precise exception control is difficult!
Exceptions must behave as you would expect! Requires tight coupling of hardware/software

Hardware contract:

- Stop offending instruction
- Let all prior instructions complete
- Flush any preceding instructions
- Set a register to show the cause
- Save the address of offending instruction(s)
- Jump to prearranged address
Multiple Exceptions

Exceptions must behave as you would expect! Requires tight coupling of hardware/software.

Operating System (software) contract:

- Look at cause of instruction (in cause register)
- Act appropriately to the cause
  - e.g. undefined instruction = kill program and return indicator of reason
  - I/O device request or OS call = save state of program, perform task, restore program to continue execution
Exceptions are why the ability to save and restore the state of any task is critical.
Instruction-Level Parallelism
Instruction-Level Parallelism (ILP) – Multiple instructions executing simultaneously

Pipelining is a form of ILP

How do you increase ILP?
Instruction-Level Parallelism (ILP) – Multiple instructions executing simultaneously

Pipelining is a form of ILP

How do you increase ILP?

- Increase depth of the pipeline
- Multiple issue = Duplicate components
Instruction-Level Parallelism

What is the goal of ILP?
Reduce Clocks-per-instruction (CPI)
Modern processors have CPI of $\frac{1}{3}$ to $\frac{1}{6}$

*Often use IPC instead so numbers get bigger*
Laundry analogy:
Assume washing machine takes 3x time of every other stage

Unbalanced pipeline – components are in busy wait
Cycle time = 1.5 hours
Extending the Pipeline

Laundry analogy:
Use 3 different machines to do portions of washer? (e.g. Wash, rinse, spin)

<table>
<thead>
<tr>
<th>Time</th>
<th>6 PM</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>1</th>
<th>2 AM</th>
</tr>
</thead>
</table>

Task order:

- A
- B
- C
- D

Use 3 different washers with counter & 3/1 Mux
Extending the Pipeline

General idea – extend pipeline to balance stages until overhead of larger pipeline $>$ performance gain

Where does the overhead come from?
Extending the Pipeline

General idea – extend pipeline to balance stages until overhead of larger pipeline > performance gain

Where does the overhead come from?
Hazards!
Structural hazards = only X stages can access any given component at a given time
Data hazards = More instructions in pipeline mean additional forwarding or stalls
Control hazards = Deeper pipeline before control decision can be made = more stalls
Modern CPU Architectures have between 14-20 stage pipelines

<table>
<thead>
<tr>
<th>Year</th>
<th>Micro-architecture</th>
<th>Pipeline stages</th>
<th>Max Clock [MHz]</th>
<th>Tech process [nm]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>Airmont (die shrink)</td>
<td>14–17 (16–19 with fetch/retire)</td>
<td>2640</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Skylake</td>
<td>14 (16 with fetch/retire)</td>
<td>4200</td>
<td></td>
</tr>
<tr>
<td>2016</td>
<td>Goldmont</td>
<td>20 unified with branch prediction</td>
<td>2800</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>Kaby Lake</td>
<td>14 (16 with fetch/retire)</td>
<td>4500</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Coffee Lake</td>
<td></td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Goldmont Plus</td>
<td>? 20 unified with branch prediction ?</td>
<td>2800</td>
<td></td>
</tr>
<tr>
<td>2018</td>
<td>Cannon Lake (die shrink?)</td>
<td></td>
<td>3200</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Whiskey Lake</td>
<td></td>
<td>4800</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Amber Lake</td>
<td></td>
<td>4200</td>
<td>14</td>
</tr>
<tr>
<td>2019</td>
<td>Cascade Lake</td>
<td></td>
<td>4400</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Comet Lake</td>
<td></td>
<td>4900</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sunny Cove (Ice Lake)</td>
<td>14–20</td>
<td>4100</td>
<td></td>
</tr>
</tbody>
</table>
Multiple Issue – Launch multiple instructions in one clock cycle

Requires multiple copies of all hardware
PC & PC + 4 both issued
Increment PC by 8
What could go wrong?
Multiple Issue

Multiple Issue – Launch multiple instructions in one clock cycle

Requires multiple copies of all hardware
PC & PC + 4 both issued
Increment PC by 8
What could go wrong?
Hazards!
Two forms of multiple-issue processor

Based on what component responsible for resolving hazards:

- **Static multiple issue** – Decisions made by compiler before execution
- **Dynamic multiple issue** – Combination of compiler and dynamic execution by processor
What strategies exist for preventing/resolving hazards in multiple issue?

- **Instruction packing** – Combining instructions into issue slots
  - **Issue slots** – Positions from which instructions could issue in a given clock cycles
- **Scheduling/Forwarding Control/Data hazards**
Speculation – A prediction type approach, allow processor to “guess” control decisions to begin execution of instructions early

How?
Compiler can reorder instructions
Static Example:
1. Move load instruction before branch
2. Load instruction completes while branch evaluating
3. If branch prediction correct, no stall
4. Fix up if branch incorrect to recover (overhead)
Speculation

Dynamic Example:

1. Hardware looks ahead for instructions to pre-execute
2. Store results in a buffer until needed
3. Flush buffers on incorrect speculation (overhead)
Speculation and Exceptions

What if exception occurs because of speculation?
e.g. Speculative load before null-pointer check

- Static approach – Add ISA support for deferring exceptions until control determined, & back out if incorrect control
- Dynamic approach – Buffer exception until instruction completion & flush for incorrect control speculation
Static Multiple Issue

How to perform static multiple issue?
Compiler groups instructions into “issue packets”

- Group of instructions that can be issued in a single cycle
- Determined by pipeline resources that are required

Very Long Instruction Word (VLIW)

- Specify multiple concurrent operations
- Issue packet = a very long instruction (e.g. 64 bits = 2 instructions)
Compiler must remove some/most/all hazards depending on ISA

- Reorder instructions into issue packets
- No dependencies *within* a packet
- Possible some dependencies between packets in pipeline
- Pad with noop if necessary
Two issue packets

- One ALU/branch instruction
- One load/store instruction
- 64-bit aligned
  - ALU/branch paired with load/store
  - Pad unused instruction with noop
Example: Static Dual Issue Datapath
What is the concern with dual issue?
More instructions in parallel = ?
Hazards in Dual-Issue

What is the concern with dual issue?
More instructions in parallel = More potential data/control/structural hazards!

Example: EX Data Hazard

- Forwarding avoided stalls with single issue
- Now we can’t use ALU result in load/store in same packet!
- Must split into two packets = a stall (unless can be rescheduled)

Load-use Hazard?
Hazards in Dual-Issue

What is the concern with dual issue?
More instructions in parallel = More potential data/control/structural hazards!

Example: EX Data Hazard

- Forwarding avoided stalls with single issue
- Now we can’t use ALU result in load/store in same packet!
- Must split into two packets = a stall (unless can be rescheduled)

Load-use Hazard – Still one cycle use latency, but loses 2 instructions in overhead
What are the hazards in the following code?

Loop:

lw $t0, 0($s1)  # $t0=array element
addu $t0, $t0, $s2  # add scalar in $s2
sw $t0, 0($s1)  # store result
addi $s1, $s1,-4  # decrement pointer
bne $s1, $zero, Loop  # branch $s1!=0
What are the hazards in the following code?

Loop:

```
lw   $t0, 0($s1)    # $t0=array element
addu $t0, $t0, $s2  # add scalar in $s2
sw   $t0, 0($s1)    # store result
addi $s1, $s1,-4    # decrement pointer
bne  $s1, $zero, Loop  # branch $s1!=0
```

First three instructions have data dependency ($t0)
Last two instructions have data dependency ($s1)
## Dual Issue Scheduling Example

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>addi $s1, $s1,-4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t0, 4($s1)</td>
<td>4</td>
</tr>
</tbody>
</table>

Instructions Per Clock (IPC) = \( \frac{5}{4} = 1.25 \)

Peak IPC = 2
Loop Unrolling – Another static technique, replicate loop body to expose additional parallelism

Why?
Loop Unrolling

Loop Unrolling – Another static technique, replicate loop body to expose additional parallelism

Why?
Reduce loop-control overhead!

Use different registers per replication

- Register Renaming
- Avoid loop-carried “anti-dependencies”
  - Store followed by a load of same register
  - A.k.a. “name dependence” = reuse of a register name
### Loop Unrolling Example

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loop:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s1, $s1,-16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>5</td>
</tr>
<tr>
<td>addu $t3, $t4, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td>nop</td>
<td>sw $t2, 8($s1)</td>
<td>7</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>8</td>
</tr>
</tbody>
</table>

**IPC = 14/8 = 1.75!**

Closer to 2, but at cost of registers & code size
Dynamic Multiple Issue = “superscalar” processors
CPU determines whether to issue 0, 1, 2, ..., instructions each cycle

- Avoids structural and data hazards
- Avoids need for compiler scheduling (but it still helps!)
Dynamic Pipeline Scheduling

Allow CPU to execute instructions out of order to avoid stalls!
Example:

```
lw $t0, 20($s2)
addu $t1, $t0, $t2
sub $s4, $s4, $t3
slti $t5, $s4, 20
```

Sub instruction can start with lw while addu is waiting
Requires that instruction write back in order!
Reservation stations & reorder buffer provide effective register renaming

When instruction issued to reservation station:

- If operand available in reg. file or reorder buffer:
  - Copied to reservation station
  - No longer required in register! can be overwritten

- If operand not available:
  - It will be provided to reservation station by a function unit
  - Register update may not be required
Dynamic Scheduling & Speculation

Predict branch $ continue issuing to reservation stations
Don’t commit until branch outcome determined

Load Speculation – Avoid load & cache miss delay

• Predict effective address
• Predict loaded value
• Load before completing outstanding stores
• Bypass stored values to load unit

Don’t commit load until speculation cleared
Why not let compiler reschedule all code?
Not all stalls are predictable
e.g. Cache misses

Can’t always schedule around branches
Different ISA implementations have different latencies & hazards
Yes!, but not as much as we would like

- Programs have dependencies that limit ILP
- Some dependencies hard to eliminate
- Some parallelism hard to expose
- Memory delays and limited bandwidth make keeping pipeline full hard
- Speculation can help!
Real Stuff
Complexity of dynamic scheduling and speculation requires power
More simpler cores may be better!

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>UltraSparc III</td>
<td>2003</td>
<td>1950MHz</td>
<td>14</td>
<td>4</td>
<td>No</td>
<td>1</td>
<td>90W</td>
</tr>
<tr>
<td>UltraSparc T1</td>
<td>2005</td>
<td>1200MHz</td>
<td>6</td>
<td>1</td>
<td>No</td>
<td>8</td>
<td>70W</td>
</tr>
</tbody>
</table>
# Cortex A8 and Intel i7

<table>
<thead>
<tr>
<th>Processor</th>
<th>ARM A8</th>
<th>Intel Core i7 920</th>
</tr>
</thead>
<tbody>
<tr>
<td>Market</td>
<td>Personal Mobile Device</td>
<td>Server, cloud</td>
</tr>
<tr>
<td>Thermal design power</td>
<td>2 Watts</td>
<td>130 Watts</td>
</tr>
<tr>
<td>Clock rate</td>
<td>1 GHz</td>
<td>2.66 GHz</td>
</tr>
<tr>
<td>Cores/Chip</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Floating point?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiple issue?</td>
<td>Dynamic</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Peak instructions/clock cycle</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Pipeline stages</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Pipeline schedule</td>
<td>Static in-order</td>
<td>Dynamic out-of-order with speculation</td>
</tr>
<tr>
<td>Branch prediction</td>
<td>2-level</td>
<td>2-level</td>
</tr>
<tr>
<td>1\textsuperscript{st} level caches/core</td>
<td>32 KiB I, 32 KiB D</td>
<td>32 KiB I, 32 KiB D</td>
</tr>
<tr>
<td>2\textsuperscript{nd} level caches/core</td>
<td>128-1024 KiB</td>
<td>256 KiB</td>
</tr>
<tr>
<td>3\textsuperscript{rd} level caches (shared)</td>
<td>-</td>
<td>2- 8 MB</td>
</tr>
</tbody>
</table>
ARM Cortex-A8 Pipeline

Branch mispredict penalty = 13 cycles
Core i7 Pipeline
Core i7 Performance
Fallacies and Pitfalls

Fallacy 1 – Pipelining is easy

- The *idea* is easy
- The implementation can have a lot of complexity

Fallacy 2 – Pipelining ideas are independent of technology

- While true in principle, dependent on # of transistors
- e.g. Deep branch prediction/Dynamic Execution requires many transistors
Fallacies and Pitfalls

Pitfall 1 – Failure to consider ISA adversely affects pipeline

- Complex instruction set (e.g. IA-32 (Pentium processors))
  - Significant overhead to make pipelining work
  - Led to IA-32 micro-operation approach
- Complex addressing modes create overhead
- Delayed branches
  - Advanced pipelines have long delay slots
1. ISA influences design of datapath and control
2. Datapath and control also influence the design of the ISA
3. Pipelining improves instruction throughput (not latency!) using parallelism
4. Hazards!
5. Instruction Level Parallelism
   • Multiple Issue
   • Dynamic Scheduling