Gradual Introduction of OOO Execution

<table>
<thead>
<tr>
<th>Frontend</th>
<th>Issue</th>
<th>Unblock</th>
<th>Commit</th>
<th>Pipelines/Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>Fixed latency pipelines, scoreboard</td>
</tr>
<tr>
<td>1201</td>
<td>10</td>
<td>10</td>
<td>000</td>
<td>Scoreboard</td>
</tr>
<tr>
<td>1205</td>
<td>10</td>
<td>10</td>
<td>000</td>
<td>Scoreboard, reorder buffer + issue queue</td>
</tr>
<tr>
<td>103</td>
<td>10</td>
<td>000</td>
<td>000</td>
<td>Scoreboard + issue queue</td>
</tr>
<tr>
<td>1025</td>
<td>10</td>
<td>000</td>
<td>000</td>
<td>Scoreboard + issue queue, reorder buffer + issue queue</td>
</tr>
</tbody>
</table>

Example Code Sequence

```
0  MUL  r1, r2, r3
1  ADDIU r11, r10, 1
2  MUL  r5, r1, r4
3  MUL  r7, r5, r6
4  ADDIU r11, r11, 1
5  ADDIU r13, r12, 1
6  ADDIU r14, r14, 2
```

To simplify discussion, all designs are single issue.

Two independent sequences of instructions offer flexibility in terms of how we schedule these instructions into a total order.

We can do the scheduling statically in software or dynamically in the hardware.
IV: Wonder front-end, issue, writeback, commit

Following is basic wonder pipeline, single issue

Split X|M stages into two functional units. Not strictly necessary but helps incremental design, single issue

What if we want to incorporate a four-cycle pipelined integer multiplier? Extend all pipelines so equal width

To avoid increasing CPI, need full bypassing which can be expensive. To help, add new issue stage when necessary. Is read and instruction is "issued" to appropriate functional unit.
ARF = Architectural Register File
SB = Scoreboard

Use scoreboard + centralize structural + data hazard checks

Scoreboard data structure

Data avail?
P = Pending; write to dest in flight
FU = Which functional unit is writing this register

Data avail? = Where is the write data in the FU pipeline

A one in DATA avail? in col i means input data is in stage i of functional unit FU.

Can use FU and DATA avail? fields to determine when to bypass and where to bypass from.

A one in column zero means that cycle that functional unit is in the current stage.

Bits in DATA avail field shift to the right every cycle.
Previous data structure not quite correct. What about now? Scoreboard should support more than one 1 per row, but with only a single FU field we don't know where to bypass from.

Solution? FU bits should be in DATA avail? Keys + shift appropriately.

Diagram on next page is still fine, since no way and just really tracking valid bits.
I202: IN ORDER FRONTEND ISSUE, OUT OF ORDER WRITEBACK/COMMIT

Scoreboard is similar to before except now we can use it to track structural hazards on the writeback path.

Set bit in DATA_AVAI according to when writeback will happen.

Because this microarchitecture conservatively avoids WAW hazards by stalling if decode scoreboard does not need FU bits in every entry of when avail? field, i.e., only one bit will be set per row in scoreboard.
**What if instruction 0 causes an exception?** If there are no exceptions, it is important to make sure that earlier instructions are done.

```plaintext
0  F D I Y 0 Y 1 Y 2 Y 3
1  F D I X W
2  F D I Y 0 Y 1 Y 2 Y 3
3  F D I Y 0 Y 1 Y 2 Y 3
4  F D I Y 0 Y 1 Y 2 Y 3
5  F D I Y 0 Y 1 Y 2 Y 3
6  F D I Y 0 Y 1 Y 2 Y 3
7  F D I Y 0 Y 1 Y 2 Y 3
8  F D I Y 0 Y 1 Y 2 Y 3
9  F D I Y 0 Y 1 Y 2 Y 3
10 F D I Y 0 Y 1 Y 2 Y 3
11 F D I Y 0 Y 1 Y 2 Y 3
12 F D I Y 0 Y 1 Y 2 Y 3
13 F D I Y 0 Y 1 Y 2 Y 3
14 F D I Y 0 Y 1 Y 2 Y 3
15 F D I Y 0 Y 1 Y 2 Y 3
16 F D I Y 0 Y 1 Y 2 Y 3
```

**Value from after exception**

**Already committed!**

**Squash unused instructions**

Commit point needs to be in MI roughly due to stores, but an early commit point prevents certain kinds of exceptions. Definitely a possibility though.
I20I: Worder Frontend Issue, ODO Writeback, Inorder Commit

![Diagram of processor stages]

<table>
<thead>
<tr>
<th>ARF</th>
<th>SB</th>
<th>RB/W</th>
<th>PAF</th>
<th>ROB</th>
<th>MAF</th>
<th>FSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>R/W</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
</tbody>
</table>

Start seeing multiple stages needing to read/write (usually of different operations) at same time.

* Significant complexity to ensure no structural hazards and that data structures are functionally correct

PAF = Physical Register File (sometimes called future file in my context)
ROB = Reference Buffer
FSB = Finished Store Buffer (1 entry)
Reorder Buffer (ROB)

- State = free, pending, finished
- S = speculative
- St = store bit
- V = physical reg specific valid
- Preg = physical reg specific

Managed like a queue, circular buffer.

- Next instruction allocates new in D
- Tail of ROB
- Speculative because branch is in flight
- Entry wrote ROB out of order
- Head of ROB
- Commit stage is waiting for head of ROB to be finished

Finished Store Buffer (FSB)

- V op Addr Data

Only one entry because only one reg instr in flight at a time. This also means don't really need to "Alloc" entry in FSB.
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19
0  MUL  r1, r2, r3
  ADDIW  r10, r11, 1
1  MUL  r3, r4
2  MUL  r5, r6
3  ADD  r12, r13, r14
4  ADD  r15, r16
5  ADD  r17, r18
6  ADD  r19, r20

Univack to puts reg to
commit to arm reg to
physical reg
the

ROB

0123

entry = free entry in ROB

state of ROB at beginning of cycle!

Pending entry in ROB

line = φ change

Circle = finished (cycle in W)

Last cycle before entry is freed
(cycle in C stage)

Entry becomes finished and
is freed on next cycle.
WHAT IF FIRST INSTRUCTION CAUSES AN EXCETION?

\[ \times F D I \times W C \]  \[ \times F D I \times W C \]  \[ \times F D I \times W C \]  \[ + F F F ] \[ \text{VALUE NOT COMMITTED YET!} \]

\[ \times F D I \times I Y o / \]  \[ + F D I \times I Y o / \]  \[ \times F D D D I \]  \[ + F F F ] \[ \text{NEED TO COPY VALUES FROM ARC TO PAR (MAY NEED MULTIPLE CYCLES)} \]

WHAT ABOUT BRANCHES?

\[ \text{on } F D I \times W C \]
\[ \text{opA } F D I \times W C \]
\[ \text{opB } F D I \times W C \]
\[ \text{opC } F D I \times W C \]

\[ \text{SQUASH SPECULATIVE INSTR + CLEAR NOF SUFFIXES} \]

Above scenario requires squashing instructions in the pipeline. Can also simply wait for all speculative instructions to reach commit stage and squash then:

\[ \text{br } F D I \times W C \]
\[ \text{opA } F D I \times W C \]
\[ \text{opB } F D I \times W C \]
\[ \text{opC } F D I \times W C \]

So more possible schemes in decreasing complexity based on when we squash speculative instr + NOF entries:

1. As soon as branch resolves
2. When branch commits
3. Gradually as speculative instructions reach commit

They are ordered in decreasing performance. If we only use one speculative bit, we can only have a single branch in flight. A second branch will need to stall to decode (structural hazard or speculative bit). The longer we wait to squash speculative instructions the worse our structural hazard.

We can also add extra speculative bits to enable multiple speculative branches in flight at once.
Can we avoid stalling entire pipeline or stale miss?

---

**Pipeline Stages:**

- **Read:** Read operation
- **Write:** Write operation
- **Commit:** Commit operation

**Op Codes:**
- **A:** FDI X WC
- **B:** FDI SW C C C
- **C:** FDI X W W W W C
- **D:** FDI I I I I X C

**Stall Reasons:**
- **Stage B:** Stage B is stalled due to a cache miss.
- **Stage C:** Stage C stalls with B-stage decoupling.
- **Stage D:** Stage D stalls with C-stage decoupling, allowing the pipeline to continue.

**Order:** Order commit only committed stages use R stage.
Revisiting Branches

I4

1. MUL (r4, r5, 1)
2. ADDW (r7, r8, r9, 1)
3. ADDU (r10, r11, 1)
4. ADDU (r12, r13, 1)

FD I Y0 Y1 Y2 Y3 W
FD I Y0 Y1 Y2 Y3 W
FD I Y0 Y1 Y2 Y3 W
FD I Y0 Y1 Y2 Y3 W

No spec instr can writeback

I202

1. MUL
2. ADDW (r7, r8, r9, 1)
3. ADDU (r10, r11, 1)
4. ADDU (r12, r13, 1)

FD I Y0 Y1 Y2 Y3 W
FD I Y0 Y1 Y2 Y3 W
FD I Y0 Y1 Y2 Y3 W
FD I Y0 Y1 Y2 Y3 W

No spec instr can writeback

I201

1. MUL
2. ADDW (r7, r8, r9, 1)
3. ADDU (r10, r11, 1)
4. ADDU (r12, r13, 1)

FD I Y0 Y1 Y2 Y3 W
FD I Y0 Y1 Y2 Y3 W
FD I Y0 Y1 Y2 Y3 W
FD I Y0 Y1 Y2 Y3 W

No spec instr can writeback

Need to squash instr in pipeline before branch
So they don't write PAF. Can either deallocate
in nos immediately or wait until commit
IO3: Modern frontend, and core issue, writeback, commit

```
F - D - I
```

```
ACK
5A
IRQ
W
```

 ISSUE QUEUE (IQR)

```
<table>
<thead>
<tr>
<th>op</th>
<th>imm</th>
<th>V</th>
<th>P</th>
<th>sel0</th>
<th>P</th>
<th>sel1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<--- TAIL pointer
<--- HEAD pointer
```

```
op = opcode
imm = immediate value
V = valid (instruction was corresponding, dest, src)
P = pending (waiting on operands to be available)
```

```
instruction ready = (V \text{src} \equiv \text{II} \text{src}) \& \& (V \text{src} \equiv \text{II} \text{src})
```

Add no structural hazards

Add need in factor in oversizing for high performance
Centralized vs. Distributed Issue Queues

**Centralized**

```
D     IQ2
    /     \
X      Y0
    \     /
10     11
  \     /
   S
```

**Distributed**

```
D     IQ2     Y0
    \     \     /     /
    IQA   IA    X
    /     \     /     /
  10     11    12
    \     /     /
     S
```
Front-end runs ahead, issues out-of-order. Decoded via IQ. IQ means present.

Assume all instructions in issue queue:

0: mult $r1, $r2, $r3
1: add $r6, $r10, $r1
2: mult $r5, $r1, $r4
3: mult $r3, $r1, $r6
4: add $r13, $r1, $r1
5: add $r13, $r1, $r1
6: add $r13, $r1, $r1

Any performance benefit vs. pipeline diagram at top?
IO2I: INORDER FRONTEND, 000 ISSUE/WRITEDACK, INORDER COMMIT

USE CENTRALIZED RESOURCES
SB, ROB, IQ
AT BEGINNING & END OF PIPELINE
PARALLEL DISTRIBUTED CONTROL
IN MIDDLE OF PIPELINE

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 mul r1, r2, r3
1 addu r1, r1, r0
2 mul r5, r1, r4
3 mul r7, r5, r6
4 addu r12, r11, r1
5 addu r14, r13, r1
6 addu r16, r15, r2
So cannot support control speculation with I03, not just implies exceptions but must stall on return.

Treat branch misprediction like exception. Have to copy ASR to PAF.

Could have bit vector at front of pipeline that says when to get register value from PAF or ASR. This would avoid need to copy ASR on mispredicted recovery.

Cannot return control flow early because save with below branched may not have committed yet. If we copy ASR into PAF early then we may end up with inconsistent PAF - we need to get the precise (imprecise) view of the ASR.

We can't just return control flow early and use the PAF because the PAF may have incorrect, speculative values.
**Example for ISD IW** where neglecting control flow IX works fine.

A: `ADDU R1, R2, 1`    FD IX WC
B: `ADDU R3, R1, R4`    FD IX WC
C: `BNE R5, R0, D`    FD IX WC
D: `ADDU R6, R1, 1`    FD IX WC

---

<table>
<thead>
<tr>
<th>Cycle</th>
<th>D I WC</th>
<th>Pref</th>
<th>Arch</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>10 16</td>
<td>10 16</td>
</tr>
<tr>
<td>1</td>
<td>BA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>10 16</td>
<td>10 16</td>
</tr>
<tr>
<td>6</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

This is the correct architectural state we would expect if we execute this code on an ideal single cycle processor.
EXAMPLE FOR J02I WHERE INSTRUCTING CONTROL FLOW IS X
DOES NOT WORK + PRODUCES INCORRECT EXECUTION

A  ADDU r1, r1, 1  0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
B  ADDU r3, r1, 1  1  0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
C  BNE r1, r0, D  3  4  5  6  7  8  9 10  11  12  13  14  15  0  1  2  3  4
D  ADDU rB, r1, 1  4  5  6  7  8  9  10  11  12  13  14  15

<table>
<thead>
<tr>
<th>Cycle</th>
<th>D IN C</th>
<th>PNF</th>
<th>ANF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td></td>
<td>1 4 7 10 13 16</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A  C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>B  C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>D  C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>D  C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>D  C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C  B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>C  B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>B  C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>B  C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>B  C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>B  C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>C  B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>C  B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>C  B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

INCORRECT RESULTS AFTER INSTRUCTION D

KEY ISSUE IS WE OVERWROTE A VALUE IN THE PNF FROM THE ART TOO SOON - BEFORE THAT UPDATE
COULD BE APPLIED.
These notes are meant to complement the lectures on out-of-order execution by providing more details on the five microarchitectures:

- **I4** : in-order front-end, issue, writeback, commit
- **I2O2** : in-order front-end/issue, OOO writeback/commit
- **I2OI** : in-order front-end/issue, OOO writeback, in-order commit
- **IO3** : In-order front-end, OOO issue, writeback, commit
- **IO2I** : In-order front-end, OOO issue and writeback, in-order commit

---

**I4: in-order front-end, issue, writeback, commit**

---

- add extra stages to create fixed length pipelines
- add scoreboard (SB) between decode and issue stages
  + one entry for each architectural register
  + pending bit : register is being written by in-flight instr
  + data location shift register : when/where data available

- Pipeline
  + D: standard decode stage; handle verifying branch and jump target prediction as normal
  + I: add issue stage because of all of bypassing logic/checks; check SB to determine if source register is pending; if not pending then get value from architectural register file; if pending then check SB to see if we can bypass the value; if RAW hazard stall; issue instruction to desired functional unit; update SB
  + X/Y/M: resolve branches in X; extra stages to delay writeback until the same stage; M0 = address generation, M1 = data memory tag check and data access; additional M1 stages may be necessary on miss; miss stalls entire pipeline to avoid complexity in eventually writing back the data
  + W: remove information in SB about instr; writeback result data to architectural register file

- Hazards
  + WB structural : not possible; in-order writeback
  + Reg RAW : check SB in I; stall, bypass, read regfile in I
  + Reg WAR : not possible; in-order issue with registered operands
  + Reg WAW : not possible; in-order writeback in W
  + Mem RAW/WAR/WAW : not possible; in-order memory access in M1

- Comments
  + Microarchitecture supports precise exceptions, but the commit point is in M1 since that is where stores happen. For longer fixed length pipelines, this means the commit point might be relatively early in the pipeline.
  + The scoreboard is technically not required, all of the necessary information is already present in the pipeline registers, but it can be expensive to aggregate this information. A scoreboard centralizes control information about what is going on in the pipeline into a single, compact data structure resulting in fast access.
The scoreboard can contain other information to enable fast structural hazard checks. For example, we can track when non-pipelined functional unis are busy.

There is no issue with allowing WAW dependencies in the pipeline at the same time, since all instructions writeback in order; but this does make our scoreboard a little more sophisticated. We can have multiple valid entries in a single row of the scoreboard; multiple instructions either in the same or different functional units can be writing the same register. We give each functional unit an identifier and shift this identifier every cycle in the data location field of the scoreboard. The combination of the column and the functional unit identifier tells us if we can bypass and from where.

Fixed length pipelines simplify managing structural hazards but has two consequences depending on how we resolve RAW hazards:
- If we stall until the value is written to the register file, then the RAW delay latency becomes significant
- If we bypass, then the bypass network becomes very large.

I2O2: in-order front-end/issue, out-of-order writeback/commit

- begin with I4 microarchitecture
- make small changes to scoreboard (see comments)

Pipeline

D: to conservatively avoid WAW hazards, stall if current dest matches any dest in SB; handle verifying branch and jump target prediction as normal

I: check SB to determine if source register is pending; if not pending then get value from architectural register file; if pending then check SB to see if we can bypass the value; if RAW hazard stall; check SB to determine if there will be a writeback port conflict and if so stall; issue instruction to desired functional unit; update SB

X/Y/M: resolve branches in X; M0 = address generation, M1 = data memory tag check and data access; additional M1 stages may be necessary on miss; miss stalls entire pipeline to avoid complexity in eventually writing back the data

W: remove information in SB about instr; writeback result data to architectural register file

Hazards

+ WB structural : check SB in I; stall in I
+ Reg RAW : check SB in I; stall, bypass, read regfile in I
+ Reg WAR : not possible; in-order issue with registered operands
+ Reg WAW : check SB in D; stall in D
+ Mem RAW/WAR/WAW : not possible; in-order memory access in M1

Comments

Scoreboard is a little simpler than in the I4 microarchitecture since we are conservatively avoiding WAW hazards by stalling in decode. We can add an extra functional unit field and the data location shift register then only needs one bit per column.
Stalling in D to avoid WAW hazards is conservative, because many times we can have multiple instructions in flight that are writing to the same register without a WAW hazard. The key is just that the older instruction has to write first, and this is likely since the older instruction is already in the pipeline. We can make the scoreboard more sophisticated to track WAW hazards and allow the I stage to issue an instruction as soon as possible. Note that writeback structural hazards and WAW hazards are two completely different hazards, but it is possible to manage both of them efficiently in a clever scoreboard implementation.

Microarchitecture does not support precise exceptions, since writeback of the architectural register file happens out-of-order. Could potentially move commit point into the M1 stage and this would allow us to handle precise exceptions, but this might prevent us from being able to handle exceptions late in the pipeline. Even so this is a viable alternative.

Out-of-order writeback significantly reduces the amount of bypassing without increasing the RAW delay latency. Note that we should not expect this microarchitecture to actually perform better than the I4 microarchitecture; in fact, it may perform worse due to the extra stalls from writeback port conflicts. Eliminating the extra "empty" pipeline registers does not decrease the RAW delay latency compared to an I4 microarchitecture with bypassing. The primary advantage of the I2O2 vs I4 microarchitecture is significantly less hardware and possible a faster cycle time due to the simpler bypass network.

Note that not all instructions (e.g., branches, jumps, jump register, stores) need to reserve the writeback port.

There is no issue with out-of-order writeback with respect to committing speculative state after a branch prediction because branches are resolved in X before any other instruction could commit early.

I2O1: in-order front-end/issue, out-of-order writeback, in-order commit

- begin with I2O2 microarchitecture
- add extra C stage at end of the pipeline to commit results in-order
- add physical regfile in W stage and architectural regfile in C stage
- split M0/M1 stages into L0/L1 load pipe and separate S store pipe
- add reorder-buffer (ROB) in between W and C stages
  + state field: entry is either free, pending, or finished
  + speculative bit: speculative instruction after a branch
  + store bit: entry is for a store
  + valid bit: is destination register valid?
  + dest: destination register specifier
- add finished store buffer (FSB) in between W and C stages
  + op field: store word, halfword, byte, etc
  + address: store address
  + data: store data
- Pipeline
  + D: to conservatively avoid WAW hazards, stall if current dest matches any dest in ROB; to conservatively avoid memory dependency hazards, stall a memory instruction if another memory instruction
is already in flight (check ROB); allocate next free entry in the ROB so that that entry’s state is now pending (ROB allocation happens in-order); if this instruction has a destination value, then set valid bit on the destination register specifier as well; if ROB is full then stall; include the allocated ROB pointer as part of the instruction control information as it goes down the pipeline; if decoded instruction is a branch, set speculative bit in ROB; to simplify branch speculation, only allow one speculative branch in flight at a time (i.e., stall in D if there is a branch in D and at least one speculative bit is set in the ROB); if instruction is not a branch but at least one speculative bit is set in the ROB, set the speculative bit for this instruction as well; handle verifying branch and jump target prediction as normal

+ I: check SB to determine if source register is pending; if not pending then get value from physical register file; if pending then check SB to see if we can bypass the value; if RAW hazard stall; check SB to determine if there will be a writeback port conflict and if so stall; issue instruction to desired functional unit; update SB

+ X/Y/L/S: resolve branches in X and if taken, then immediately redirect the front-end of the pipeline, squash all instructions in-flight behind the branch right away, and free all speculative entries in the ROB, pass along whether or not the branch was mispredicted to the W stage; L0 = load address generation, L1 = load tag check and data access; miss stalls entire pipeline to avoid complexity in eventually writing back the data; S = store address generation

+ W: remove information in SB about instr; writeback result data to physical register file; use ROB pointer that has been carried with the instruction control information to change the state of corresponding entry in the ROB to finished; if store instruction set store bit in the ROB and write the finished store buffer with the store address and data

+ C: commit point; when entry at the head of the ROB (the oldest in-flight instruction) is finished; if entry has a valid write destination, use the destination register specifier in the ROB to copy the corresponding data from the physical register file into the architectural register file; if entry is a store, use the address and data in the finished store buffer to issue a write request to the memory system; if store misses, stall entire pipeline; change state of the entry at the head of the ROB to free; if entry at the head of the ROB is a mispredicted branch, advancing the head pointer to the next non-free entry since there will be freed entries after the branch due to the misprediction

- Hazards
  + WB structural : check SB in I; stall in I
  + ROB structural : stall in D if full
  + Reg RAW : check SB in I; stall, bypass, read physical regfile in I
  + Reg WAR : not possible; in-order issue with registered operands
  + Reg WAW : check ROB/SB in D; stall in D
  + Mem RAW/WAR/WAW : check ROB/SB in D; stall so 1 mem instr in-flight
- Comments

+ Technically, we can check for WAW hazards, check for in-flight memory/branch instructions, and allocate an ROB entry in the I stage (since this is in-order issue), but to be consistent with the later microarchitectures we do these operations in the D stage.

+ Note that even if an instruction does not writeback a result (e.g., branches, jumps, jump register, stores) we still need to reserve the writeback port so that we can update the ROB and possibly the FSB.

+ Microarchitecture supports precise exceptions, since commit happens in-order in the C stage. When processing an exception in the C stage, we must squash all instructions in the pipeline, including instructions that are either pending or finished in the ROB. Then we need to copy the values in the architectural regfile into the physical register file. Finally, we can redirect the control flow to the exception handler.

+ Branches are handled with the speculative bit in the ROB. Currently, we can only handle a single speculative branch in flight a time, but we could add additional speculative bits to enable multiple speculative branches to be in-flight at once.

+ To avoid stalling the entire pipeline on store misses, we can add an R stage after the C stage with a second store buffer in between the R and C stage. This store buffer is called the committed store buffer (CSB) and is sometimes physically unified with the finished store buffer (FSB). The C stage now does not access memory but instead simply moves committed stores into the CSB. The R stage can now take its time sending those store operations into the memory system. When the store has actually updated memory we say it has "retired". Note that this is after the commit point, so once a store is in the CSB it must eventually write to memory. We can monitor the FSB and CSB when implementing memory fences to ensure memory consistency. We will probably also need some kind of store acknowledgement which may mean entries in the CSB have three states: free, committed, retiring (wait for store ack).

------------------------------------------------------------------
IO3: In-order front-end, out-of-order issue, writeback, commit
------------------------------------------------------------------
- begin with I2O2 microarchitecture

- add issue queue (IQ) in between D and I stages
  + opcode : opcode for instruction
  + immediate : immediate if necessary
  + valid bit, dest : register destination specifier
  + valid bit, pending bit, src 0 : reg source 0 specifier
  + valid bit, pending bit, src 1 : reg source 1 specifier

- Pipeline

  + D: to conservatively avoid WAW hazards; stall if current dest matches any dest in IQ or SB; to conservatively avoid WAR hazards, stall if current dest matches any sources in IQ; if IQ is full then stall; to conservatively avoid memory dependency hazards, stall a memory instruction if another memory instruction is already in flight (check SB); this microarchitecture cannot allow speculative instructions to issue (see notes), so we stall in decode if any
branch is currently unresolved in the pipeline; handle verifying branch and jump target prediction as normal

+ I: consider only "ready" instructions in IQ; to determine if an instruction is ready: (1) for all valid sources check if the pending bit in the IQ is not set, (2) for pending valid sources in the IQ check the SB to see if we can bypass the source, (3) check the SB to determine if there will be a writeback port conflict. So a ready instruction has ready sources (either in the register file or available for bypassing) and has no structural hazards; choose oldest ready instruction to issue meaning we do the register read or bypass, remove it from the IQ, and send it to the correct functional unit; update SB

+ X/Y/M: resolve branches in X, if branch prediction was correct then instructions stalled in D will be allowed to proceed, if branch prediction was incorrect then squash instructions in F and D an redirect the front of the pipeline; M0 = address generation, M1 = data memory tag check and data access; additional M1 stages may be necessary on miss; miss stalls just the memory pipeline (although only one memory instruction can be in-flight at a time anyways); load/store does not writeback until miss has been processed

+ W: remove information in SB about instr; writeback result data to architectural register file

- Hazards
  + WB structural : check SB in I; stall in I
  + IQ structural : stall in D if full
  + Reg RAW : check IQ/SB in I; stall, bypass, read regfile in I
  + Reg WAR : check IQ in D; stall in D
  + Reg WAW : check IQ/SB in D; stall in D
  + Mem RAW/WAR/WAW : check IQ/SB in D; stall so 1 mem instr in-flight

- Comments
  + This microarchitecture has a centralized IQ. A common alternative is to use a distributed IQ organization, where we partition the IQ so that each partition issues to a disjoint subset of the functional units. The D stage enqueues the instruction into the appropriate IQ partition in-order. We can issue instructions out of each partition out-of-order. A distributed IQ reduces implementation complexity, although it can also reduce performance compared to the centralized IQ.

  + This microarchitecture does not support precise exceptions, since writeback of the architectural register file happens out-of-order. We cannot move the commit point earlier into the pipeline because issue is out-of-order, and obviously we cannot move the commit point into the D stage.

  + A key disadvantage of this microarchitecture is that it cannot support issuing speculative instructions out-of-order with respect to a pending branch. The problem is that these speculative instructions may writeback to the physical register file, and there is no way to recover the original state if we determine later that the branch was mispredicted.

  + Note that not all instructions (e.g., branches, jumps, jump register, stores) need to reserve the writeback port
IO2I: In-order front-end, OOO issue and writeback, in-order commit

- begin with I2O2 and combine I2O1 and I03 microarchitectures
- add extra C stage at end of the pipeline to commit results in-order
- add physical regfile in W stage and architectural regfile in C stage
- split M0/M1 stages into L0/L1 load pipe and separate S store pipe
- add reorder-buffer (ROB) in between W and C stages
- add finished store buffer (FSB) in between W and C stages
- add issue queue (IQ) in between D and I stages

Pipeline

+ D: to conservatively avoid WAW hazards; stall if current dest matches any dest in ROB; to conservatively avoid WAR hazards, stall if current dest matches any sources in IQ; if IQ is full then stall; to conservatively avoid memory dependency hazards, stall a memory instruction if another memory instruction is already in flight (check ROB); allocate next free entry in the ROB so that that entry’s state is now pending (ROB allocation happens in-order); if this instruction has a destination value, then set valid bit on the destination register specifier as well; if ROB is full then stall; include the allocated ROB pointer as part of the instruction control information as it goes down the pipeline; if decoded instruction is a branch, set speculative bit in ROB; to simplify branch speculation, only allow one speculative branch in flight at a time; if instruction is not a branch but at least one speculative bit is set in the ROB, set the speculative bit for this instruction as well; handle verifying branch and jump target prediction as normal

+ I: consider only "ready" instructions in IQ; to determine if an instruction is ready: (1) for all valid sources check if the pending bit in the IQ is not set, (2) for pending valid sources in the IQ check the SB to see if we can bypass the source, (3) check the SB to determine if there will be a writeback port conflict. So a ready instruction has ready sources (either in the register file or available for bypassing) and has no structural hazards; choose oldest ready instruction to issue meaning we do the register read from the PRF or bypass, remove it from the IQ, and send it to the correct functional unit; update SB

+ X/Y/L/S: resolve branches in X and pass whether prediction was correct or not to W stage; L0 = load address generation, L1 = load tag check and data access; miss stalls entire pipeline to avoid complexity in eventually writing back the data; S = store address generation

+ W: remove information in SB about instr; writeback result data to physical register file; use ROB pointer that has been carried with the instruction control information to change the state of corresponding entry in the ROB to finished; if store instruction set store bit in the ROB and write the finished store buffer with the store address and data

+ C: commit point; when entry at the head of the ROB (the oldest in-flight instruction) is finished; if entry has a valid write destination, use the destination register specifier in the ROB to copy the corresponding data from the physical register file into the architectural register file; if entry is a store, use the
address and data in the finished store buffer to issue a write request to the memory system; if store misses, stall entire pipeline; change state of the entry at the head of the ROB to free; if committed instruction is a mispredicted branch, copy the ARF into the PRF, redirect control flow, change state of speculative entries in the ROB to free, and squash corresponding instructions in pipeline; if branch is correctly predicted remove speculative bit for all entries in the ROB

- Hazards
  + WB structural : check SB in I; stall in I
  + IQ structural : stall in D if full
  + ROB structural : stall in D if full
  + Reg RAW : check IQ/SB in I; stall, bypass, read PRF in I
  + Reg WAR : check IQ in D; stall in D
  + Reg WAW : check ROB/SB in D; stall in D
  + Mem RAW/WAR/WAW : check ROB/SB in D; stall so 1 mem instr in-flight

- Comments
  + Both the IQ and ROB have some duplicated information, so some microarchitectures will unify the IQ and ROB into a single structure (often just called an ROB). An instruction lives in the unified IQ/ROB from just after it is decoded until it is committed. Each instruction in the unified ROB goes through various states: pending, ready, executing, finished, retiring, etc.
  + This microarchitecture supports precise exceptions since commit happens in-order. On an exception, we must copy the ARF into the PRF before redirecting the control flow. Possible to add a committed store buffer as in the I2OI microarchitecture.
  + Note that we cannot naively redirect the control flow early on a mispredicted branch early as we could do in the microarchitectures with in-order issue. Assume we redirect control flow in X and immediately copy the ARF into the PRF; an instruction from way before the branch may still be in flight and it might overwrite an entry in the PRF which contains a value from an instruction just before the branch that wrote back before the branch resolved. In other words, we can’t immediate copy the ARF into the PRF in X because the ARF does not contain a "precise" view of the architectural state of the machine right after the branch until all instructions before the branch of committed. Similarly, we cannot redirect the control flow in X and use the PRF since the PRF can contain speculative state from squashed instructions. We have no option but to wait until all instructions from before the branch commit; then we can redirect the control flow. Note that modern machines use more sophisticated data-structures that leverage features already required for register renaming to allow early redirection of the control flow.
  + We could add per-register bits in the the I stage to track whether or not the value in the PRF is valid. If the bit is set, then we read from the PRF, if the bit is not set then we read from the ARF. This would help avoid having to immediate copy the ARF into the PRF on an exception or (more importantly) branch mispredict -- but of course requires the ARF to have more read ports.
  + Note that even if an instruction does not writeback a result (e.g., branches, jumps, jump register, stores) we still need to reserve the writeback port so that we can update the ROB and possibly the FSB.