WAW and WAR "NAME" Dependencies

WAW|WAR dependencies are NOT "true" DATA dependencies. Raw
dependencies are "true" DATA dependencies because we actually
pass DATA from the writer to the reader.

NAME dependencies just exist because we have a limited
number of "names" (registers, specifiers or memory addresses).

We will update IX
on cycle 9 for r4,
meaning instruction 2
could get the wrong
value for r4.

We will commit the wrong
value here since instruction
1 overwrote r4. This will
cause imprecise exceptions
and prevent speculation on
branches.

Current IO2I Microarchitecture: Conservatively Stalls

Stall in decode on any potential
WAW|WAR hazard.

What if we could use MORE architectural registers?

Adding more "names" removes the NAME dependency, but
our arch NAME space is limited.

Register renaming = rename registers in hardware so WAW|WAR

REGISTER RENAMING OVERVIEW

2 SCHEMES
- POINTERS IN IQ | ROB
- VALUES IN IQ | ROB

IO2I was pointers in IQ and ROB so we will start there. By adding reg renaming to IO2I.

Reg renaming with points in IQ and previous in ROB

All data structures are the same as IO2I
- Except add two fields to ROB
- Add rename task (AT) and free list (FL)
- Increase the size of the PAF

This will give us more "names" to use!
**Modified Reorder Buffer (ROB)**

- **State**: Free, Pending, Finished
- **S**: Speculative
- **S+**: Store bit
- **V**: Rest is valid
- **Preg**: Physical reg spec
- **Arej**: Arch reg spec
- **Pprej**: Previous physical reg

**Rename Table (RT)**

- **Arch reg specifier**
  - P: Pending bit
  - Preg - What physical register is this arch register mapped to?

**Free List (FL)**

- If bit is zero, physical register is free and can be used for renaming.
- If bit is one, physical register is already in use and cannot be used for renaming.
<table>
<thead>
<tr>
<th></th>
<th>MUL</th>
<th>R1, R2, R3</th>
<th>FD i - I X 1 X WC</th>
<th>MUL</th>
<th>R4, R5</th>
<th>FD i - I X 1 X WC</th>
<th>ADD</th>
<th>R2, R4, R1</th>
<th>FD i - I X 1 X WC</th>
<th>ADD</th>
<th>R2, R4, R1</th>
<th>FD i - I X 1 X WC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>RUC</th>
<th>R1, R2, R3, R4, R5</th>
<th>FL</th>
<th></th>
<th>IQ</th>
<th></th>
<th>R0S</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

---

who can we free physical req?*

who next write to map co architectural regular commit
Freeing Physical Registers

Addu $r1, $r2, $r3  ← Assume arch $r1 mapped to preg $p0
Addu $r4, $r1, $r5
Addu $r1, $r6, $r7
Addu $r8, $r9, $r10

0 Addu $r1, $r2, $r3  ← $I_x(WC)
1 Addu $r4, $r1, $r5
2 Addu $r1, $r6, $r7
3 Addu $r8, $r9, $r10

Write $p0  Free $p0  Alloc $p0  Write $p0  Read $wrong
Value in $p3!

0 Addu $r1, $r2, $r3  ← $I_x(WC)
1 Addu $r4, $r1, $r5
2 Addu $r1, $r6, $r7
3 Addu $r8, $r9, $r10

Write $p0  Alloc $p2  Write $p2  Dealloc $p0

If arch $r3 is mapped to preg $p3, we can Free $p3 when
the next instruction that writes $r3 commits

Unified Physical | Arch Register File

Combine PIF + AIF into one large register file.
Replace AIF w/ architectural rename table
Instead of copying values, commit stage copies
preg pointer into appropriate entry of arch rename table
Unified preg/areg can be smaller than preg + areg separate
REG RENAMING with values in IQ + 12 ROB

Similar to previous design except:
- modified ROB
- modifies RT
- no FL
**Modified ROB**

```
OP  IMM  S  V  Dist  V.P  SRC0  V.P  SRC1
```

```
# Stone value in ROB instead of pointer to PVP
```

**Modified IQ**

```
if reading, source field contains index into ROB - this is kind of like a preg specific
```

**Modified_rename_table**

```
V  P  Preg
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

V = valid bit
- if `0` then value in AAIf is up to date
- if `1` then value is either in flight or in ROB

P = reading bit
- if `0` then value in ROB
- if `1` then value is in flight

Preg = index into ROB
can free "physical register when " instructions

which writes that "physical register commut.

All consumers have value in IQ or can get

value from ACE.

-20.5 -20.5 45 = 0.5
0 -20.5 2.0 -3.0
0 2.0 -6.0 3.0
0 -6.0 3.0
0 3.0
0 -20.5

-20.5 -20.5 -23.0 -20.5 2.0 -3.0
0 -20.5 2.0 -3.0
0 2.0 -6.0 3.0
0 -6.0 3.0
0 3.0
0 -20.5

-20.5 -20.5 -23.0 -20.5 2.0 -3.0
0 -20.5 2.0 -3.0
0 2.0 -6.0 3.0
0 -6.0 3.0
0 3.0
0 -20.5

-20.5 -20.5 -23.0 -20.5 2.0 -3.0
0 -20.5 2.0 -3.0
0 2.0 -6.0 3.0
0 -6.0 3.0
0 3.0
0 -20.5

-20.5 -20.5 -23.0 -20.5 2.0 -3.0
0 -20.5 2.0 -3.0
0 2.0 -6.0 3.0
0 -6.0 3.0
0 3.0
0 -20.5

-20.5 -20.5 -23.0 -20.5 2.0 -3.0
0 -20.5 2.0 -3.0
0 2.0 -6.0 3.0
0 -6.0 3.0
0 3.0
0 -20.5
MEMORY DISAMBIGUATION

Register dependencies are preserved. Register renaming now eliminate RAW and RAW Register "name" dependencies.

What about RAW, RAW, WAR MEMORY DEPENDENCIES?

Previous microarchitectures either execute all memory instructions in same stage or only allows one mem instruction in flight at a time.

Our 000 Mach exec/loads/stores in different stages (L1 vs. C)

How do we get multiple mem instr in flight at once?

EXAMINE 3 MICROARCHITECTURES

1. In order load/store issue + unified stores
2. In order load/store issue + split stores
3. Out-of-order load/store issue

IN ORDER LOAD/STORE ISSUE

RAW HAZARDS?

\[
\begin{align*}
\text{sw } r1, 0(r2) & \quad F D I S W C \\
\text{sw } r3, 0(r4) & \quad F D I S W C \\
\text{assume } r[r2] = r[r4] \\
\end{align*}
\]

Because all stores exec in order w C stage, there are no RAW hazards.

WAR HAZARDS?

\[
\begin{align*}
\text{lw } r1, 0(r2) & \quad F D I S U W C \\
\text{sw } r3, 0(r4) & \quad F D I S W C \\
\text{assume } r[r2] = r[r4] \\
\end{align*}
\]

Because store cannot exec before earlier load, there are no WAR hazards.

This assumes once a load reaches L1, it goes into the memory system and a store to the same address cannot bypass it.
RAW HAZARDS?

```
sw r1, 0(r2)
lw r3, 0(r4)
FDI SW F
FDI LOW F C
```

Assume \( a[r2] = a[r4] \) in order load/store issue

IN ORDER LOAD/STORE ISSUE VS. UNIFIED STORES

- Integer EQ supports 000 issue
- LOAD EQ only supports 10 issue
- MAKE FSB DEEPER, ALLOCATE 1001 IN D STAGE
- SEARCH ADDRESSES IN FSB IN D STAGE (SEARCH FSB AS WELL)
  - If match then RAW dependency exists of weight instr
    1) Stall
    2) BYPASS STORE DATA TO LOAD (LAST STORE IN PROJ ORDER)
  - If no match then no RAW dependency exists of weight instr
    (ET LOAD) ACCESS MEMORY AND WRITE BACK
- If spilling, new address search can be conservative to simplify hardware
- "If by-passing, then address search must be exact"
**Simple Example**

```plaintext
sw r1, 0(r2)           F D I S W F C
lw r3, 0(r4)           F D I [L] W F C
```

(\(\text{w} \leftrightarrow \text{search FS3 + DURING STORE DATA}
\)
\(\text{notice that we must DURING ADDRESS}
\)
\(\text{check since address} \phi \text{ written until}
\)
\(\text{end of cycle} \)

**More complicated example**

```plaintext
lw r1, 0(r2)           F D I (U) W C
lw r3, 0(r4)           F D I (U) W C
mv r5, r1, r3          F D I (T) y W C
lw r6, 0(r6)           F D I (T) S W C
lw r7, 0(r8)           F D I [I] W C
```

(\(\text{assume}
\)
\(\text{r(r8)} \neq r(r6)\)

\(\text{load is stuck TABLED store due to}
\)
\(\text{INDRN ISSUE even though no aliasing.}
\)

\(\text{* BUT we know truth earlier!} \)

**IN ORDER LOAD / STORE ISSUE \& SPLIT STORES**

**In Decode for store**

1) **IF STORE DATA IS READY** (IE. \text{pending})
   - THEN ENQUEUE STORE \& LOAD STORE ISSUE QUEUE
     \text{AS NORMAL}

2) **IF STORE DATA IS NOT READY** (IE. \text{pending})
   - THEN SPLIT STORE INTO TWO PARTS
     - STORE DATA PART GOES IN INTEGER IQ \text{(use \text{x PIPE})}
     - STORE ADDRESS PART GOES IN \text{x DIST IQ}

**In Writeback for Split store**

- STORE DATA PART UPDATES DATA IN FS3 + SETS VALID bit IN FS3
- STORE ADDR PART UPDATES ADDR IN FS3 + SETS VALID bit IN FS3

\(\text{once both valid bits set, the change state to finished}
\)
\(\text{for STORE in T03}
\)

(\(\text{* might need to stall load if store data} \phi \text{valid} \))
Aside: Memory Queues + Load Misses

Even with memory queues or load cache misses, still do not need to worry about WAR hazards.

Stores happen in commit, commits happen in order, so load must commit before store, and before load commits it must load the data.
**OUT-OF-ORDER LOAD/STORE ISSUE**

_WILL HAZARDS?

Still no will hazards due to inorder commit.

_WILL HAZARDS?

\[
\begin{align*}
& \text{l} \text{w} \ r_1, \ o(\text{r}2) & \text{FD} & \text{i} & \text{I (W)} & \text{D} & \text{C} \\
& \text{l} \text{w} \ r_3, \ o(\text{r}4) & \text{FD} & \text{I} & \text{SW} & \text{r} & \text{C}
\end{align*}
\]

Still not a problem. See previous page.

_WILL HAZARDS?

\[
\begin{align*}
& \text{l} \text{w} \ r_1, \ o(\text{r}2) & \text{FD} & \text{i} & \text{ISUC} \\
& \text{l} \text{w} \ r_3, \ o(\text{r}4) & \text{FD} & \text{I (W)} & \text{lr} & \text{C}
\end{align*}
\]

If R[r2] = R[r4], then checking FSB in 50 will not help because addres for store is not in FSB yet!

Store is still in IQ waiting to issue.

**SPECULATION**

Speculatively issue load assuming no raw hazard. Check later to see if raw hazard has occurred, and if so, test squash all instructions after load and restart execution at load instruction.

When finished, load store load address in finished load buffer until commit, allocate FLS and work in D.
How do we check for raw hazards?

1. Check FLB when store in 5 stage
2. Check FLB when store commits
3. Re-execute load when it commits + compare value

000 lost issue of check in 5 stage

Store searches in FLB for the store address. If store address matches a load address which is "younger" than store, a RAW has occurred. Mark load + restart execution when load commits.

Raw dep issued in-order:

sw r1, 0(r2)  F D I S W r  C
lw r3, 0(r4)  F D i

Raw potentially bypass this data, but need to make sure only bypass from younger store that is older than load!

Raw dep issued out-of-order:

sw r1, 0(r2)  F D i  I S W r  C
lw r3, 0(r4)  F D i  I O U W r
add r5, r3, 1
add r6, r5, 1

Detect RAW hazard, restart execution at (1a)

War dep issued in-order:

sw r1, 0(r2)  F D I  I O U W r  C
sw r3, 0(r4)  F D i

This load is older than store, should not cause RAW hazard.

War dep issued out-of-order:

lw r1, 0(r2)  F D i  I O U W r  C
sw r3, 0(r4)  F D i S W r  C

Search FLB and FWD no match, no hazards.
More complicated example:

0  SW r1, 0(r2)  FD IS WR  C
1  SW r3, 0(r4)  FD i  FDI is WR  / FDI i0  FDI i
2  LW r5, 0(r6)  FD i  FDI iW C  / FDI i
3  SW r7, 0(r8)  FD i  FDI iW R  / FDI i
4  ADDI r9, r5, 1  FD i  FDI iW
5  ADDI r10, r9, 1

Assume R[r2], R[r4], R[r6], R[r8] are all equal.

Detect raw hazard, mark load as violation, restart execution after load.

OOO load issue with check in C stage:

Store searches FLD for store address, in C stage, on match mark load as violation.

Raw dep issued in-order:

SW r1, 0(r2)  FD IS WR  C
lw r3, 0(r4)  FD i  FDI iW R  C

Bypass? Raw hazard?

Raw dep issued out-of-order:

SW r1, 0(r2)  FD i  FDI 0U WR  C
lw r3, 0(r4)

Match so restart execution at load.

Checking in commit is simple, but difficult to integrate with bypassing.
000 1st issue w/ RE-EXECUTE LOAD AT COMMIT

When committing load re-execute load + compare data to
what was retrieved in L1. It does not match then restart
execution at load.

RAW DEP ISSUED IN-ORDER

sw r1, 0(r2)  FDI JSR c
lw r3, 0(r4)  FDI I = w,r (c)

MATCH, NO VIOLATION

RAW DEP ISSUED OUT-OF-ORDER

sw r1, 0(r2)  FDI I = w,r (c)
lw r3, 0(r4)  FDI I = w,r (c)

NOT MATCH, VIOLATION

SIMPLE, BUT REQUIRES EXTRA MEMORY REQUEST TBW.

OPTIMIZATIONS

- Only re-execute instructions that depend on the load
- Unify FSD + FIB to simplify ordering queues
- Can still potentially split stores to resolve raw violations
- Predict when violations will occur, + base on this
  prediction, stall load if it is predicted to depend
  on earlier store.

ASIDE: SPECULATION

Once we have a general mechanism for speculation +
recovery via PONS + (SNAPSHOT REMAP TABLES), can use
for a variety of optimizations

- Branch prediction
- Memory RAW HAZARD PREDICTION
- LOAD HIT PREDICTION
- LOAD VALUE PREDICTION
- Memory consistency prediction