ECE 511: Computer Architecture 6. Oct. 2004

Lecture 12: ILP Semantics (Why the Reorder Buffer Works)

Lecturer: Matt Frank *Scribe: Eric Zimmerman*

Goals:

To pipeline deeply
To execute instructions out-of-order

Cautions - WAW (output dependence)

```
R7 <- MULT 3, 4
R7 <- ADD 19, 20
…
Read R7
```

1. <u>If we fail to enforce output dependencies, then we may get a bad result.</u>

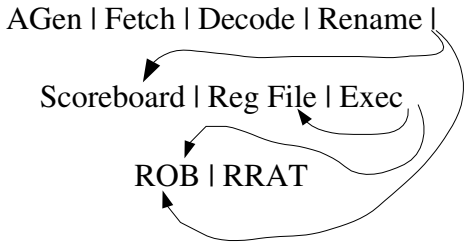
How do we *know* we handled all the cases of dependencies? This can be proven formally.

We want to guarantee correct executions for all sequences of code, but we cannot enumerate all sequences of code.

Why is what we've done so far sufficient? We know it is necessary that (1). But do we *know* ROB mechanism is sufficient to produce correct results?

What do we mean by "correct"? We have a set of semantics (meaning) for our ISA. This is formally defined in instruction set manual in Register Transfer notation. Every instruction does some register transfer (assuming in-order execution).

For a machine start state, we can map the states at the end of ROB+RF+RRAT concretely into states of Sequential model.

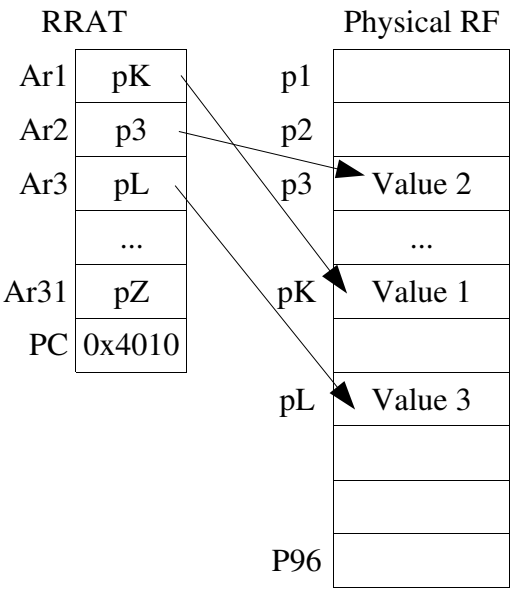


Claim: At retirement, the PC of instruction retiring matches exactly the retirement PC of the sequential model.

Proof technique: inductive

Basically: If we are in a "reasonable" state, all transitions are also to reasonable states.
How do we know we start in a reasonable state? We build the machine that way!

Mapping diagram: We map these values to those in the sequential register file.



We measure the state of the sequential model at the end of each clock edge.
At the end of each cycle, either

- an instruction retires
- an instruction does not retire

For sequential model state, there are 32 registers + PC = 33*64 bits of state
Out-of-order model has a much larger state (a superset of sequential model)

We are arguing correctness is achieved at retirement time, not otherwise.

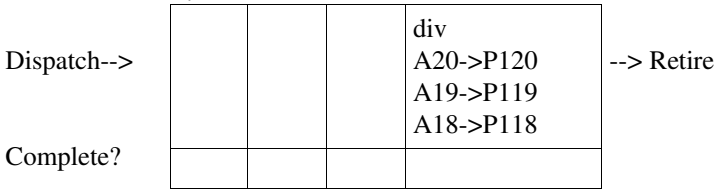
If we had no branch instructions, renaming and scheduling rules take care of matching sequential model. With branch instructions, every instruction depends on the sequence of branches before it.

```
…
      beq  r17, target
      mul r20, r23-> r19
…
target:
```

Whether or not we write r19 in the multiplication instruction depends on branch immediately before.

ROB Structure

retiring `div r20 <- r19, r18`



By inductive hypothesis, All prior instructions preserved correct instruction semantics.

How do we know A18->P118 and A19->P119? It is because the rename stage works in order! Since rename is in order and instructions proceed sequentially with respect to Rename RAT and Retirement RAT structures, it must be that the mappings are consistent with the sequential model.

There is a free list of physical registers stored in the rename table. If a physical register is in the free list, it cannot be in the RAT, SB, execution, ROB, or anywhere else in the pipeline!

After retirement, P120 contains the correct value and A20 points to it.

How do we know the ROB never stalls indefinitely? Serial rename and retirement guarantees the absence of circular dependencies.

If something retires in the ROB, the retirement RAT and Register File corresponds to sequential state transition for executing the instruction.

We can extend the ROB to deal with:

- load/store aliasing
- interrupts
- circuit errors

<u>Memory</u>

Why don't we rename memory locations?
We can't know which memory location to rename at the rename stage. The rename table would be as large as memory, imposing practical size and latency constraints.

`LD R1, 0(R2)` is equivalent to `R1 <- M[R2+0]`

Assume store goes to memory before instruction retires and architectural registers r12 and r13 are written long ago.

Scoreboard:

```
…
div r15, r16 -> r17
beq r17, target   // (mispredicted)
sw [r13] <- r12
```

Method 1: issue speculated stores to memory (bad)

Method 2: Delay issue of stores until they are at head of ROB. This causes RAW hazards with load instructions. If we delay the issue of loads…

Assume this scoreboard sequence:

```
sw [r13] <- r12
lw r15 <- [r13]
add r18 <- r15, r16
```

This will revert to sequential execution under method 2.

To be continued…