

Better Pipeline Management

EE685, Fall 2024

Hank Dietz

<http://aggregate.org/hankd/>

References

- Chapter 3 of Computer Architecture, A Quantitative Approach, 5th edition
- JILP Championship Branch Prediction, <http://www.jilp.org/cbp2014/>
- <http://nathantypanski.github.io/tomasulo-simulator/>

Remember EE380?

- Pipelined design based on single-cycle design
- Basic design issues
 - Structural hazards
 - Data dependence issues
 - Control flow dependence
- Discussed VLIW, SuperScalar, EPIC ideas, but never got into how that stuff really worked

Basic Blocks

- A **Basic Block** is a region of code which has a single entry and single exit, such that any time the block is entered, all instructions will be executed before exiting – i.e., **NO** control flow
- How big is a typical basic block?
Can be huge, often as small as **~5 instructions**
- Need to work across basic blocks to get more
Than ~5 instructions executing in parallel!

Bigger Basic Blocks?

- Compiler technology can make blocks bigger (we'll talk more about this a little later)
 - Loop unrolling, unraveling, strip mining
 - Code hoisting and other code motions
 - Trace scheduling
- Can we make hardware able to look across basic blocks *without* compiler restructuring?

Control Dependences

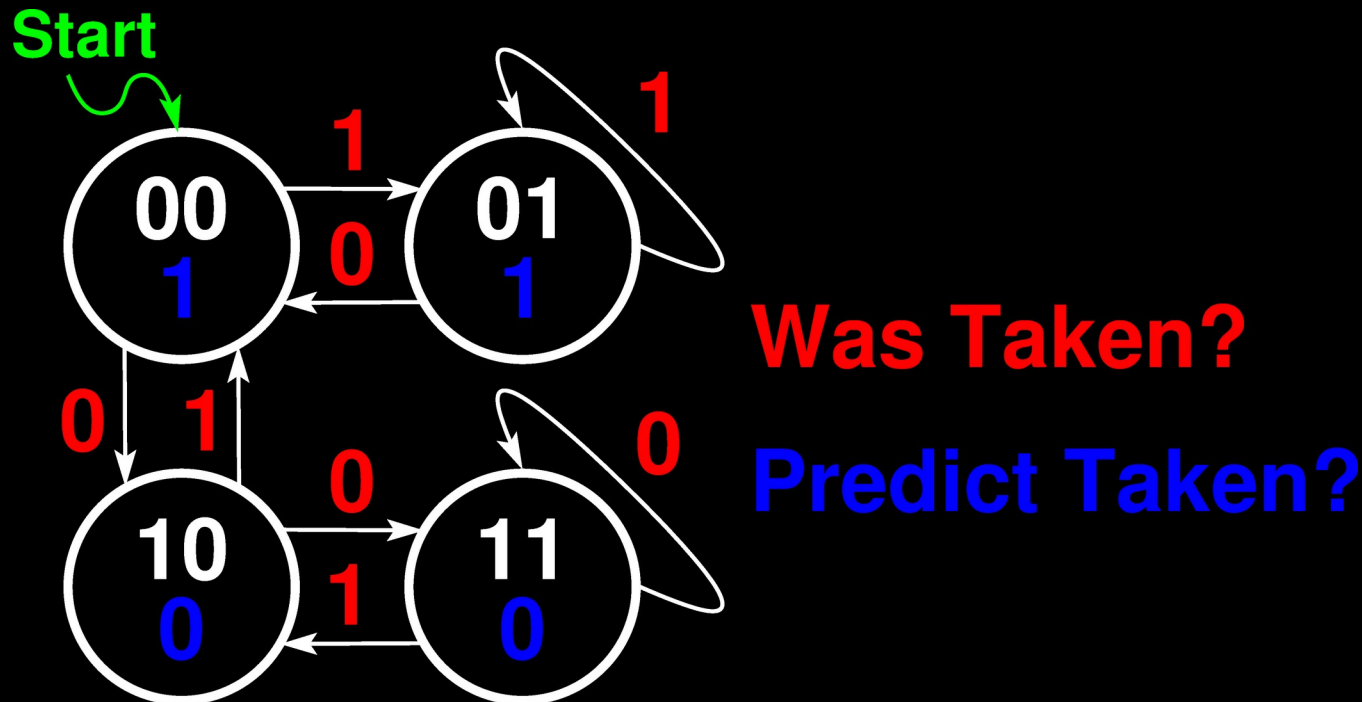
- Two flavors:
 - Branch takes an extra cycle to compute the target address from the immediate offset
 - Conditional branch/jump/skip/call/return does not decide taken/not-taken early enough
- **BTB** largely solves the first problem...
- Branch prediction tries to solve the second

Branch Prediction Methods

- Always pad with NOPs
- Predict not taken – easy, right for many forward branches such as then clause of a typical if
- Predict taken – harder (need BTB), right for backward branches in loops
- Predict forward not, backward taken
- Predict BOTH taken and not (e.g., Pentium Pro)
- Compiler marks instructions as usually taken, usually not taken, or don't know
- Use history (BHB) to predict future behavior

BHB Concepts

- Can encode either **history** or **prediction state** (remember that 4-state branch predictor?)



BHB Concepts

- Two types of history can be recorded:
 - History of *this particular branch*
(a function of address of branch instruction)
 - History of *last K branches* from anywhere
- Can have many component predictors...
 - Both types of history can be used
 - Weighted scoring of which one is best now
(i.e., a **tournament predictor**)

Other Control Flow Tricks

- **Branch folding**: both target address and a copy of the instruction there go into BTB – can save a fetch cycle
- Can remember likely indirect branch targets
- **Return address predictor**: create an internal stack to predict where return will go
- **Instruction prefetch**: issue fetches early to hide Memory latency
- Fetch blocks of instructions and extract (e.g., match idiomatic instruction sequence at a time)

Pipeline Scheduling

- **Static scheduling:**
Instructions execute in the (partial) order
Determined & specified by the compiler
- **Dynamic scheduling (out-of-order execution):**
Instructions behave as if they executed in order,
but hardware re-arranges to minimize bubbles
- Static is simpler (**precise exceptions**), etc.
- Dynamic can know about microarchitecture and
precise run-time dependencies
- **Best performance combines static + dynamic**

Von Neumann vs. Dataflow

Do instructions chase data or vice versa?

Dependence Analysis (from EE380)

- **Use or R**: reads the value bound to a name
- **Def or W**: binds a new value to a name
- **True dependence**: carries a value, $D \rightarrow U$, **RAW**
add $\$t0, \$t1, \$t2$ or $\$t3, \$t0, \$t4$
- **Anti-dependence**: kills a value, $U \leftarrow D$, **WAR**
add $\$t0, \$t1, \$t2$ or $\$t1, \$t3, \$t4$
- **Output dependence**: kills a value, $D \rightarrow D$, **WAW**
add $\$t0, \$t1, \$t2$ or $\$t0, \$t3, \$t4$

CDC 6600 Scoreboard

- Enables dynamic scheduling with both OOO execution and OOO completion
 - Instructions proceed when dep. Are met
 - WAR completion: stall W until instruction has read the operands (force program order)
 - WAW: must detect hazard & stall in decode
- ```
DIVD F0 , F2 , F4
ADDD F10 , F0 , F8
SUBD F8 , F8 , F14
```

# Scoreboard Stage 1: Issue (ID1)

- Decode instruction, check for structural hazards
  - Stall all issues if this instruction is a WAW with any W in the machine
  - If there is a free function unit, issue the Instruction there & record in scoreboard

# Scoreboard Stage 2: Read Operands (ID2)

- Wait until no data hazards, then read operands
  - Can read operand if either:
    - No RAW on the operand with W in an instruction issued earlier
    - RAW is writing now
  - Implements OOO and/or parallel execution



# Scoreboard Stage 3: Execution (EX)

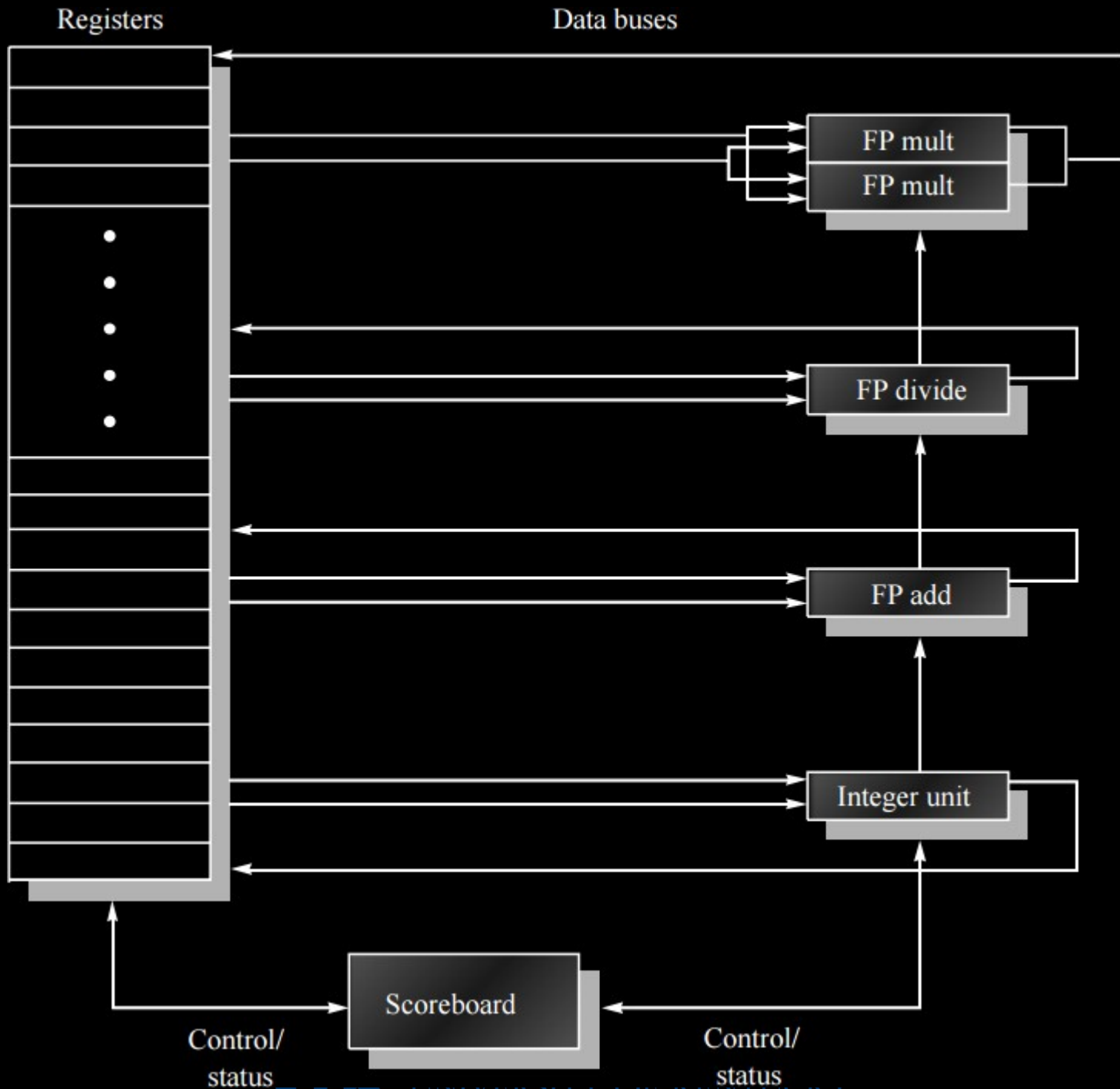
- Perform the operation on the operands
  - Notifies the scoreboard when done...

# Scoreboard Stage 3: Write Back (WB)

- Finish execution, write result back to register
  - If an earlier instruction is a RAW for this W, stall here

# Scoreboard Structure

- Instruction status: which stage is it in?
- Function unit status:
  - Busy?
  - Operation to perform (opcode)
  - Dest & Src registers:  $F_i$ ,  $F_j$ ,  $F_k$
  - Function units that produce  $F_j$ ,  $F_k$  are  $Q_j$ ,  $Q_k$
  - Are  $F_j$ ,  $F_k$  ready:  $R_j$ ,  $R_k$
- Register status: which function unit writes this?

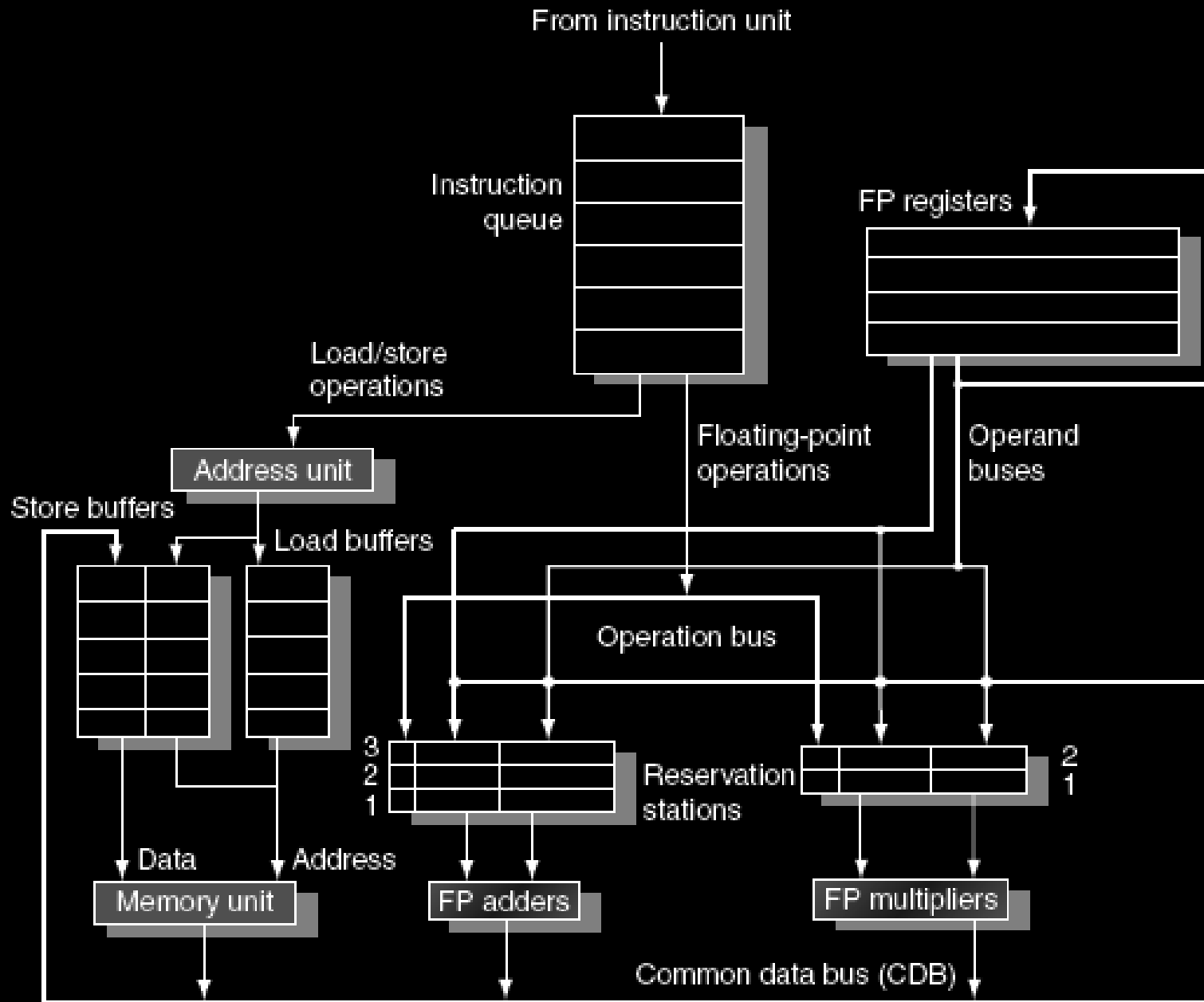


# Scoreboard Summary

- Only works one one basic block at a time
- Small number of functional units makes structural hazards common – no OOO for instructions to same function unit
- Waits for WAR hazards (after EX, before WB)
- Prevents WAW hazards (in ID)
- Still gave good speedup for CDC 6600:  
1.7X compiled code, 2.5X hand-written code

# Why **Tomasulo** Instead?

- Scoreboard is centralized;  
**Tomasulo** uses distributed **Reservation Stations**
- Reservation stations effectively implement register renaming to avoid WAR, WAW hazards
- Scoreboard must read both sources together
- **Common Data Bus (CDB)** broadcasts results to all function units
- Load and store queues are function units too



# Reservation Station Structure

- Busy?
- Operation to perform:  $Op$  (*which is not FU*)
- For each source operand  $j, k$ :
  - Reservation station producing it:  $Q_j, Q_k$
  - Ready flag:  $R_j, R_k$  (*separately ready*)
  - Value of the operand:  $V_j, V_k$
- Also tracks address for load/store
- Register result status: which FU has a pending write to each register?



# Tomasulo Stage 1: **Issue**

- Issue an instruction
  - If an RS is available, issue next instruction from FIFO instruction queue to it

# Tomasulo Stage 2: **Execute**

- Execute the operation
  - If CDB value is for one of our operands, save the value in V and set R
  - Use the FU when:
    - All earlier branches have completed
    - Load/store ordering of matching addresses would be preserved
    - All operands ( $R_j$ ,  $R_k$ ) are ready

# Tomasulo Stage 3: Write Result

- Send results everywhere they need to go
  - Write the result into CDB, which goes to:
    - Reservation stations waiting for it
    - Destination register
    - Store buffers waiting for it
  - Stores wait for both address & value

|             |           | Instruction status |         |              |
|-------------|-----------|--------------------|---------|--------------|
| Instruction |           | Issue              | Execute | Write Result |
| L.D         | F6,32(R2) | ✓                  | ✓       | ✓            |
| L.D         | F2,44(R3) | ✓                  | ✓       |              |
| MUL.D       | F0,F2,F4  | ✓                  |         |              |
| SUB.D       | F8,F2,F6  | ✓                  |         |              |
| DIV.D       | F10,F0,F6 | ✓                  |         |              |
| ADD.D       | F6,F8,F2  | ✓                  |         |              |

| Reservation stations |      |      |    |                    |       |       |               |
|----------------------|------|------|----|--------------------|-------|-------|---------------|
| Name                 | Busy | Op   | Vj | Vk                 | Qj    | Qk    | A             |
| Load1                | No   |      |    |                    |       |       |               |
| Load2                | Yes  | Load |    |                    |       |       | 44 + Regs[R3] |
| Add1                 | Yes  | SUB  |    | Mem[32 + Regs[R2]] | Load2 |       |               |
| Add2                 | Yes  | ADD  |    |                    | Add1  | Load2 |               |
| Add3                 | No   |      |    |                    |       |       |               |
| Mult1                | Yes  | MUL  |    | Regs[F4]           | Load2 |       |               |
| Mult2                | Yes  | DIV  |    | Mem[32 + Regs[R2]] | Mult1 |       |               |

| Register status |       |       |    |      |      |       |     |     |     |
|-----------------|-------|-------|----|------|------|-------|-----|-----|-----|
| Field           | F0    | F2    | F4 | F6   | F8   | F10   | F12 | ... | F30 |
| Qi              | Mult1 | Load2 |    | Add2 | Add1 | Mult2 |     |     |     |

# Tomasulo vs. Scoreboard

- Issues when RS vs. FU free
- Reads operands from CDB/reg vs. reg  
(i.e., it implements a form of reg renaming & doesn't need both operands simultaneously)
- Write values to CDB vs. reg
- WAW and WAR are not hazards
- Instructions completing/cycle 1 vs.  $k$
- Instructions start executing/cycle  $k$  vs. 1

# So, Do Modern Designs Use Scoreboard Or Tomasulo?

- No. :-)
- Many modern architectures **explicitly rename registers**... which is actually how most compilers do the equivalent analysis:
  - **SSA: Static Single Assignment**
  - Each potentially unique value gets a unique temporary (register) name

# Speculative Execution

- Allow instructions to start executing along the predicted control flow path(s) before we know
- What to do if prediction was wrong?
  - Undo the side-effects (usually hard)
  - Buffer side-effects: don't **commit** them
- Requires some extra hardware...

# Reorder Buffer (ROB)

- Where not-yet-committed things are held
- Each entry describes instruction, destination, value computed, and instruction status (completed? exception happened?)
- Tomasulo could use ROB, instead of CDB...
- Mispredict discards ROB entries
- Confirmed prediction commits
  - Writes to memory, registers (accept rename)
  - Process exceptions (precisely!)



# Multiple Issue

- Can't get  $CPI < 1$  unless  $> 1$  instruction/cycle
- Must effectively fetch  $> 1$  instruction/cycle
  - Use **VLIW/EPIC** horizontal coding  
(multiple instructions per instruction parcel)
  - Decode and group multiple individually-encoded instructions (**Superscalar**)
  - Use **SWAR** (**SIMD Within A Register**)
  - Use **LARs** (**Line Associative Registers**)
- Compiler always “stacks the deck”

| Common name               | Issue structure  | Hazard detection   | Scheduling               | Distinguishing characteristic                                       | Examples                                                                 |
|---------------------------|------------------|--------------------|--------------------------|---------------------------------------------------------------------|--------------------------------------------------------------------------|
| Superscalar (static)      | Dynamic          | Hardware           | Static                   | In-order execution                                                  | Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8 |
| Superscalar (dynamic)     | Dynamic          | Hardware           | Dynamic                  | Some out-of-order execution, but no speculation                     | None at the present                                                      |
| Superscalar (speculative) | Dynamic          | Hardware           | Dynamic with speculation | Out-of-order execution with speculation                             | Intel Core i3, i5, i7; AMD Phenom; IBM Power 7                           |
| VLIW/LIW                  | Static           | Primarily software | Static                   | All hazards determined and indicated by compiler (often implicitly) | Most examples are in signal processing, such as the TI C6x               |
| EPIC                      | Primarily static | Primarily software | Mostly static            | All hazards determined and indicated explicitly by the compiler     | Itanium                                                                  |
| SWAR                      | Static           | Primarily software | Mostly static            | Implicit by compiler                                                | MMX, SSE, ... AVX, GPUs                                                  |

# Decoupling Instruction Fetch

- Processor internals are really dataflow
- Isolate processing from instruction fetch
  - Some processors use decoupled instruction fetch engines (CSPI array processors, P4)
  - **Barrel Processing / Multithreading:** instructions don't have to come from just one PC or process (Denelcor HEP, Tera MTA, Intel Hyperthreading)
  - **SIMD virtualization** (TMC, ATI/AMD, NVIDIA)