# Lecture 18: Instruction Level Parallelism
## -- Dynamic Superscalar, Advanced Techniques, ARM Cortex-A53, and Intel Core i7

**CSCE 513 Computer Architecture**

**Department of Computer Science and Engineering**

**Yonghong Yan**

yanyh@cse.sc.edu

https://passlab.github.io/CSCE513

## Topics for Instruction Level Parallelism

- 5-stage Pipeline Extension, ILP Introduction, Compiler Techniques, and Branch Prediction
  - C.5, C.6
  - 3.1, 3.2
  - ~~Branch Prediction, C.2, 3.3~~
- Dynamic Scheduling (OOO)
  - 3.4, 3.5
- Hardware Speculation and Static Superscalar/VLIW
  - 3.6, 3.7
- Dynamic Superscalar, Advanced Techniques, ARM Cortex-A53, and Intel Core i7
  - 3.8, 3.9, 3.12
- SMT: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput
  - 3.11

# Acknowledge and Copyright

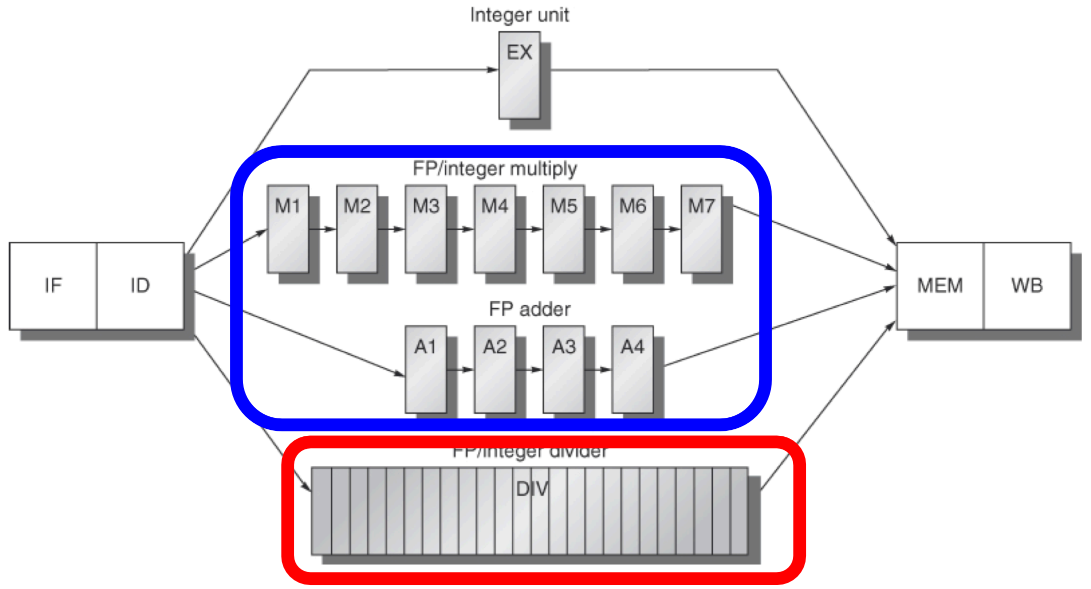
- Slides adapted from
  - UC Berkeley course “Computer Science 252: Graduate Computer Architecture” of David E. Culler Copyright(C) 2005 UCB
  - UC Berkeley course Computer Science 252, Graduate Computer Architecture Spring 2012 of John Kubiatowicz Copyright(C) 2012 UCB
  - Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley

- https://passlab.github.io/CSCE513/copyrightack.html

## Review

## Not Every Stage Takes only one Cycle

- FP EXE Stage
  - Multi-cycle Add/Mul
  - Nonpiplined for DIV



- MEM Stage

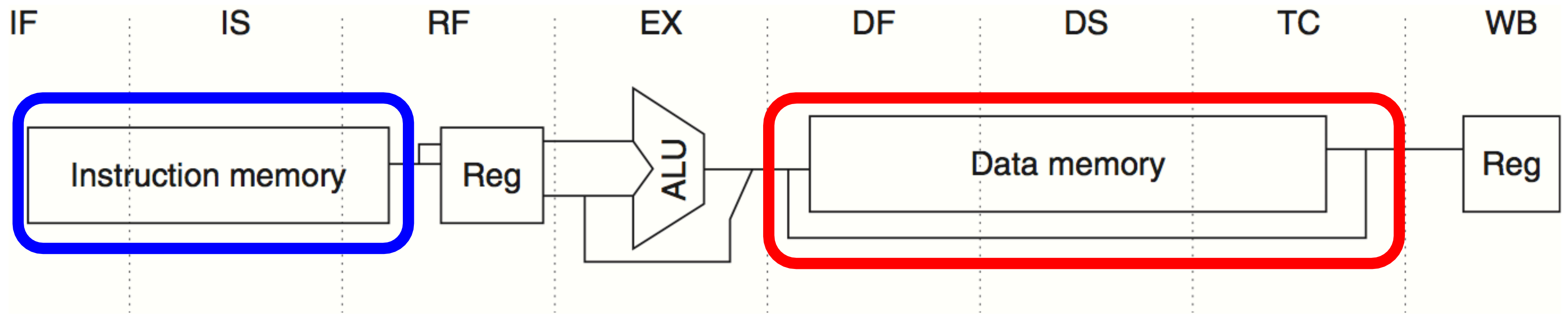


**Figure C.41 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches.** The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating

# Issues of Multi-Cycle in Some Stages

- The divide unit is not fully pipelined
  - structural hazards can occur
    - » need to be detected and stall incurred.
- The instructions have varying running times
  - the number of register writes required in a cycle can be > 1
- Instructions no longer reach WB in order
  - Write after write (WAW) hazards are possible
    - » Note that write after read (WAR) hazards are not possible, since the register reads always occur in ID.
- Instructions can complete in a different order than they were issued (out-of-order complete)
  - causing problems with exceptions
- Longer latency of operations
  - stalls for RAW hazards will be more frequent.

## Hardware Solution for Addressing Data Hazards

- Dynamic Scheduling of Instructions:
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
- Data Hazard via Register Renaming
  - Dynamic RAW hazard detection and scheduling in data-flow fashion
  - Register renaming for WRW and WRA hazard (name conflict)
- Implementations
  - Scoreboard (CDC 6600 1963)
    - » Centralized register renaming
  - Tomasulo’s Approach (IBM 360/91, 1966)
    - » Distributed control and renaming via reservation station, load/store buffer and common data bus (data+source)

# Register Renaming Summary

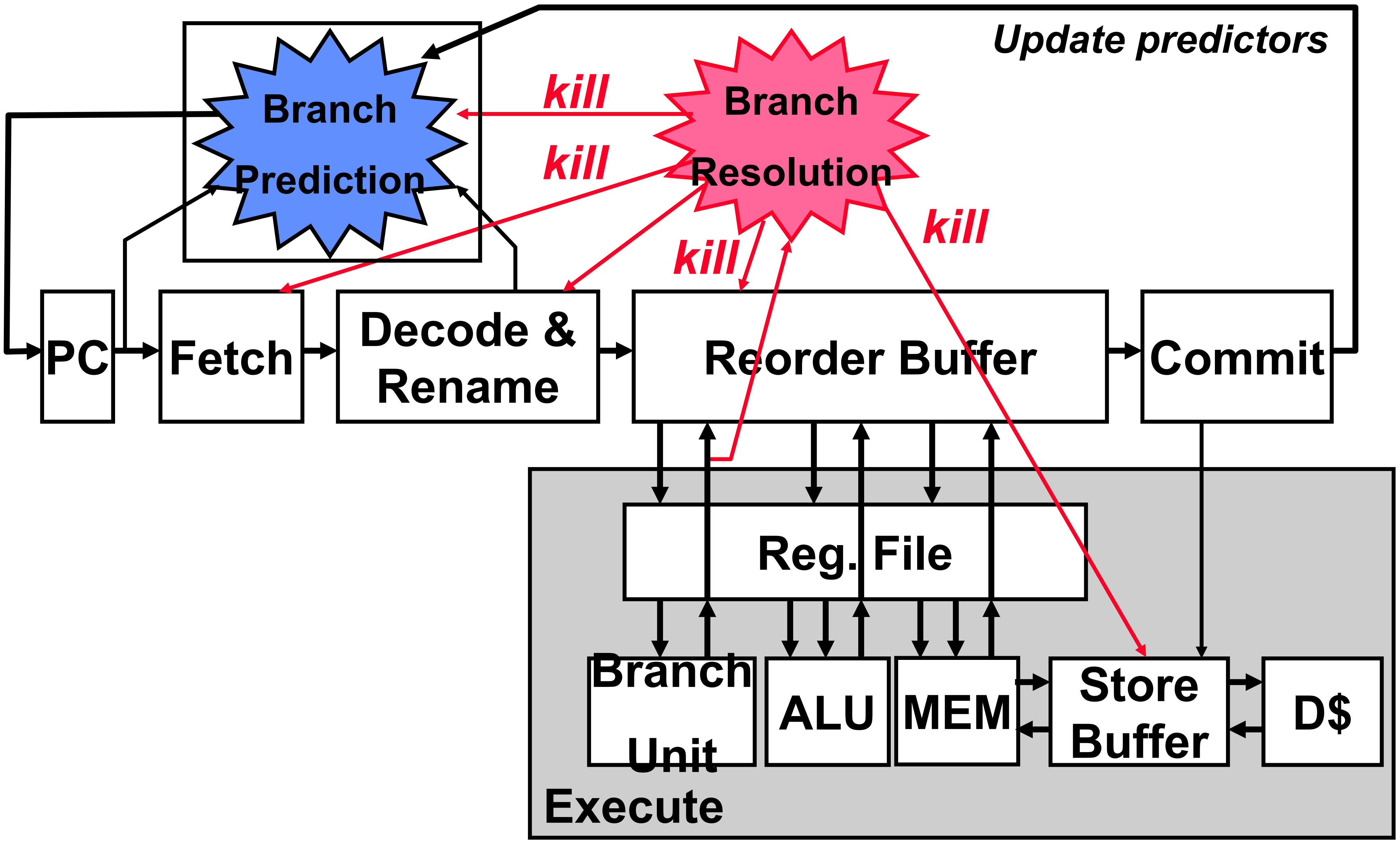
- **Purpose of Renaming: removing "Anti-dependencies"**
  - Get rid of WAR and WAW hazards, since these are not "real" dependencies
- **Implicit Renaming: i.e. Tomasulo**
  - Registers changed into values or response tags
  - We call this "implicit" because space in register file may or may not be used by results!
- **Explicit Renaming: more physical registers than needed by ISA.**
  - Rename table: tracks current association between architectural registers and physical registers
  - Uses a translation table to perform compiler-like transformation on the fly

## Hardware Speculation: Addressing Control Hazards

- **Branch Prediction:**
  - **Modern branch predictors have high accuracy: (>95%) and can reduce branch penalties significantly**
  - **Required hardware support**
    - » **Branch history tables (Taken or Not)**
    - » **Branch target buffers, etc. (Target address)**

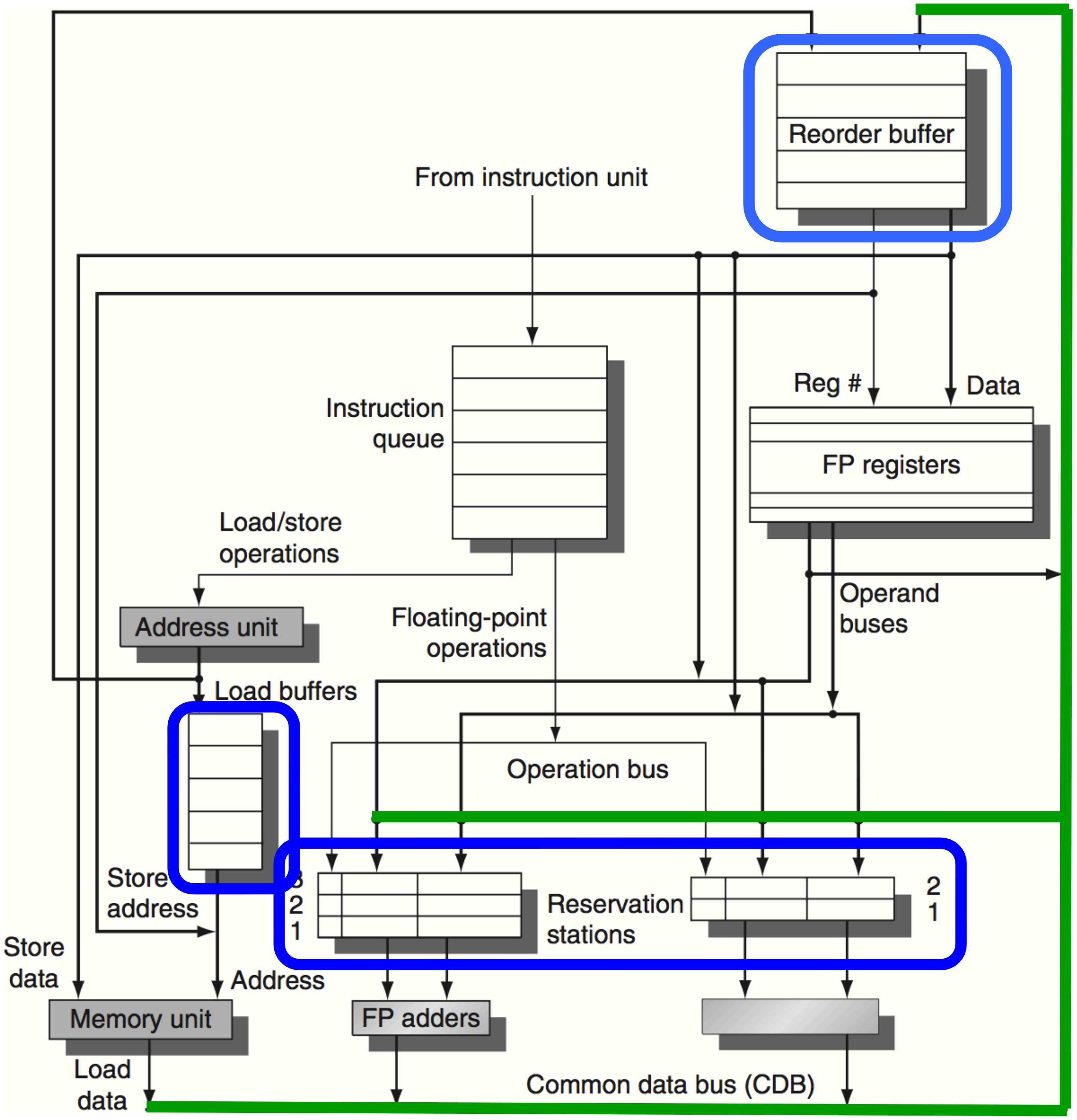
- **In-order commit for out-of-order execution:**
  - **Instructions fetched and decoded into instruction reorder buffer in-order**
  - **Execution is out-of-order ( ➔ out-of-order completion)**
  - **Commit (write-back to architectural state, i.e., regfile & memory) is in-order**

# Speculation: Prediction + Mis-prediction Recovery



# Hardware Speculation in Tomasulo Algorithm

- **Reservation Station and Load Buffer**
  - Register renaming
  - For dynamic scheduling and out-of order execution
- **Reorder Buffer**
  - Register renaming
  - For in-order commit
- **Common Data Bus**
  - Data forwarding
- **Also handle memory data hazard**



# Four Steps of *Speculative* Tomasulo

1. **Issue—get instruction from FP Op Queue**

   **If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")**

2. **Execution—operate on operands (EX)**

   **When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called "issue")**

3. **Write result—finish execution (WB)**

   **Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.**

4. **Commit—update register with reorder result**

   **When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")**

## Instruction In-order Commit

- Also called completion or graduation
- In-order commit
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
- Three cases when an instr reaches the head of ROB
  - Normal commit: when an instruction reaches the head of the ROB and its result is present in the buffer
    - » The processor updates the register with the result and removes the instruction from the ROB.
  - Committing a store:
    - » is similar except that memory is updated rather than a result register.
  - A branch with incorrect prediction
    - » indicates that the speculation was wrong.
    - » The ROB is flushed and execution is restarted at the correct successor of the branch.

# In-order Commit with Branch

```
Loop:   L.D     F0,0(R1)
        MUL.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDIU  R1,R1,#-8
        BNE     R1,R2,Loop     ;branches if R1
```

**Reorder buffer**

| Entry | Busy | Instruction | State | Destination | Value |
|---|---|---|---|---|---|
| 1 | No | L.D F0,0(R1) | Commit | F0 | Mem[0 + Regs[R1]] |
| 2 | No | MUL.D F4,F0,F2 | Commit | F4 | #1 × Regs[F2] |
| 3 | Yes | S.D F4,0(R1) | Write result | 0 + Regs[R1] | #2 |
| 4 | Yes | DADDIU R1,R1,#-8 | Write result | R1 | Regs[R1] – 8 |
| 5 | Yes | BNE R1,R2,Loop | Write result | | |
| 6 | Yes | L.D F0,0(R1) | Write result | F0 | Mem[#4] |
| 7 | Yes | MUL.D F4,F0,F2 | Write result | F4 | #6 × Regs[F2] |
| 8 | Yes | S.D F4,0(R1) | Write result | 0 + #4 | #7 |
| 9 | Yes | DADDIU R1,R1,#-8 | Write result | R1 | #4 – 8 |
| 10 | Yes | BNE R1,R2,Loop | Write result | | |

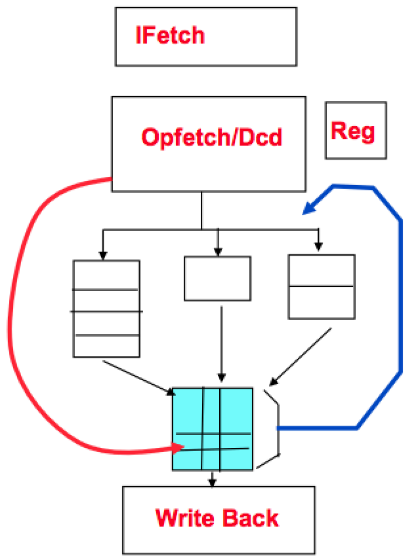
FLUSHED

IF Misprediction

**FP register status**

| Field | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 |
|---|---|---|---|---|---|---|---|---|---|
| Reorder # | 6 | | | | 7 | | | | |
| Busy | Yes | No | No | No | Yes | No | No | ... | No |

## Dynamic Scheduling and Speculation



- ILP Maximized (a restricted data-flow)
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
  - In-order commit
- Data Hazards
  - Input operands-driven dynamic scheduling for RAW hazard
  - Register renaming for handling WAR and WAW hazards
- Control Hazards (Branching, Precision Exception)
  - Branch prediction and in-order commit (speculation)
  - Branch prediction without speculation
    - » Cannot do out-of-order execution/complete for branch
- Implementation: Tomasulo
  - Reservation stations and Reorder buffer
  - Other solutions as well (scoreboard, history table)

## Multiple ISSUE via VLIW/Static Superscalar

## Textbook: CAQA 3.7

## Multiple Issue

- "Flynn bottleneck"
  - single issue performance limit is CPI = IPC = 1
  - hazards + overhead ⇒ CPI >= 1 (IPC <= 1)
  - diminishing returns from superpipelining [Hrishikesh paper!]
- Solution: issue multiple instructions per cycle

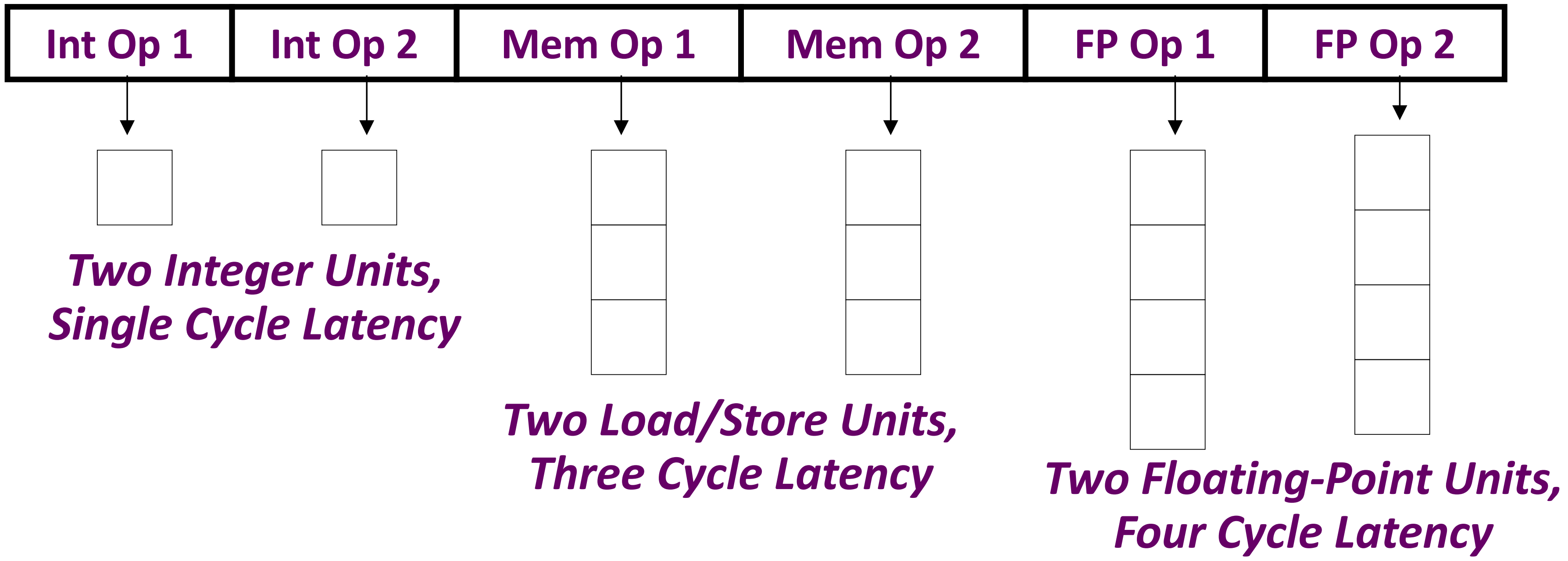
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| inst0 | F | D | X | M | W | | |
| inst1 | F | D | X | M | W | | |
| inst2 | | F | D | X | M | W | |
| inst3 | | F | D | X | M | W | |

- 1st superscalar: IBM America → RS/6000 → POWER1

## VLIW and Static Superscalar

- **Very similar in terms of the requirements for compiler and hardware support**
- **We will discuss VLIW/Static superscalar**

- **Very Long Instruction Word (VLIW)**
  - **packages the multiple operations into one very long instruction**


Int Op 1
Int Op 2
Mem Op 1
Mem Op 2
FP Op 1
FP Op 2
Two Integer Units, Single Cycle Latency
Two Load/Store Units, Three Cycle Latency
Two Floating-Point Units, Four Cycle Latency

# Recall: Unrolled Loop that Minimizes Stalls for Scalar

```
1  Loop: L.D     F0,0(R1)
2        L.D     F6,-8(R1)
3        L.D     F10,-16(R1)
4        L.D     F14,-24(R1)
5        ADD.D   F4,F0,F2
6        ADD.D   F8,F6,F2
7        ADD.D   F12,F10,F2
8        ADD.D   F16,F14,F2
9        S.D     0(R1),F4
10       S.D     -8(R1),F8
11       S.D     -16(R1),F12
12       DSUBUI  R1,R1,#32
13       BNEZ    R1,LOOP
14       S.D     8(R1),F16      ; 8-32 = -24
```

L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

| Instruction producing result | Instruction using result | Latency in clock cycles |
|---|---|---|
| FP ALU op | Another FP ALU op | 3 |
| FP ALU op | Store double | 2 |
| Load double | FP ALU op | 1 |
| Load double | Store double | 0 |

**14 clock cycles, or 3.5 per iteration**

# Loop Unrolling in VLIW

**Unrolled 7 times to avoid delays**

**7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)**

**Average: 2.5 ops per clock, 50% efficiency**

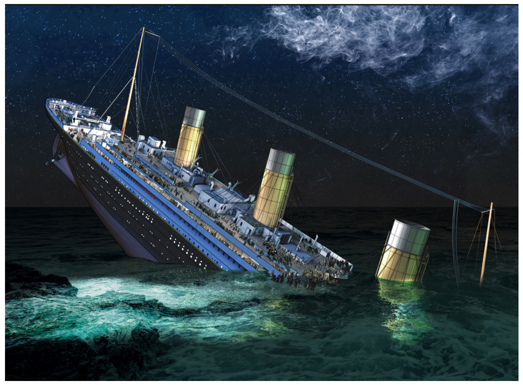
| Memory reference 1 | Memory reference 2 | FP operation 1 | FP operation 2 | Integer operation/branch |
|---|---|---|---|---|
| `L.D F0,0(R1)` | `L.D F6,-8(R1)` | | | |
| `L.D F10,-16(R1)` | `L.D F14,-24(R1)` | | | |
| `L.D F18,-32(R1)` | `L.D F22,-40(R1)` | `ADD.D F4,F0,F2` | `ADD.D F8,F6,F2` | |
| `L.D F26,-48(R1)` | | `ADD.D F12,F10,F2` | `ADD.D F16,F14,F2` | |
| | | `ADD.D F20,F18,F2` | `ADD.D F24,F22,F2` | |
| `S.D F4,0(R1)` | `S.D F8,-8(R1)` | `ADD.D F28,F26,F2` | | |
| `S.D F12,-16(R1)` | `S.D F16,-24(R1)` | | | `DADDUI R1,R1,#-56` |
| `S.D F20,24(R1)` | `S.D F24,16(R1)` | | | |
| `S.D F28,8(R1)` | | | | `BNE R1,R2,Loop` |

**Figure 3.16 VLIW instructions that** cycles assuming no branch delay; norr ations in 9 clock cycles, or 2.5 operati operation, is about 60%. To achieve thi this loop. The VLIW code sequence abo MIPS processor can use as few as two l

| Instruction producing result | Instruction using result | Latency in clock cycles |
|---|---|---|
| FP ALU op | Another FP ALU op | 3 |
| FP ALU op | Store double | 2 |
| Load double | FP ALU op | 1 |
| Load double | Store double | 0 |

## Summary

- VLIW: Explicitly Parallel, Static Superscalar
  - Requires advanced and aggressive compiler techniques
  - Trace Scheduling: Select primary “trace” to compress + fixup code
- Other aggressive techniques
  - Boosting: Moving of instructions above branches
    - » Need to make sure that you get same result (i.e. do not violate dependencies)
    - » Need to make sure that exception model is same (i.e. not unsafe)
- Itanium/EPIC/VLIW is not a breakthrough in ILP
  - If anything, it is as complex or more so than a dynamic processor
  - **Some refers to as Itanic!**
- BUT it is used today:
  - e.g. TI sigal processor C6x



# Very Important Terms

- Dynamic Scheduling → Out-of-order Execution
- Speculation → In-order Commit
- Superscalar → Multiple Issue

| Techniques | Goals | Implementation | Addressing | Approaches |
|---|---|---|---|---|
| Dynamic Scheduling | Out-of-order execution | Reservation Stations, Load/Store Buffer and CDB | Data hazards (RAW, WAW, WAR) | Register renaming |
| Speculation | In-order commit | Branch Prediction + Reorder Buffer | Control hazards | Prediction and misprediction recovery |
| Superscalar /VLIW | Multiple issue | Software and Hardware | To Increase CPI | By compiler or hardware |

## Dynamic Scheduling, Multiple Issue (Dynamic Superscalar), and Speculation

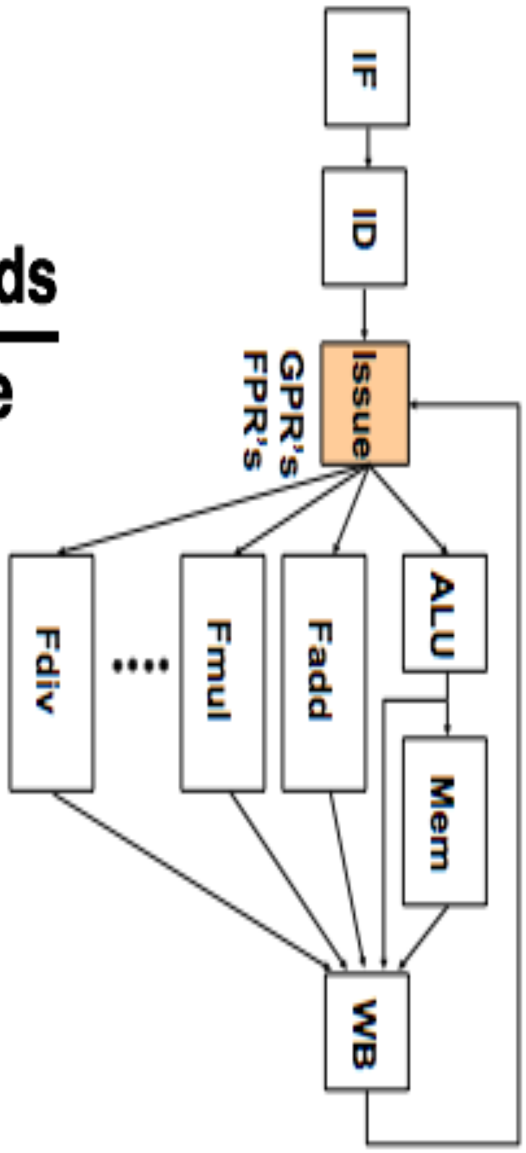
**Textbook: CAQA 3.8**

# Dynamic Scheduling, Multiple Issue, and Speculation

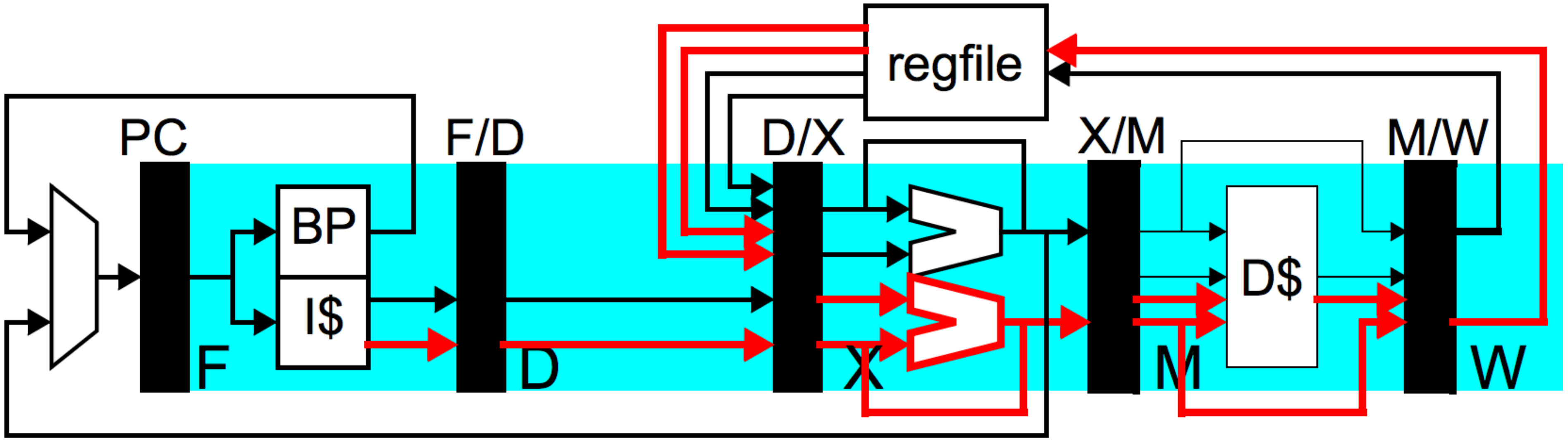
- Microarchitecture quite similar to those in modern microprocessors
  - Real

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \frac{\text{Cycles}}{\text{Instruction}} \frac{\text{Seconds}}{\text{Cycle}}$$

- Consider two issue per clock
  - Example: CPU with floating point ALUs: Issue 1 FP + 1 Integer instruction per cycle.
    - » Save at least 1 cycle than the pipeline
  - Challenges
    - » Find the right instructions
    - » Dependency between instructions



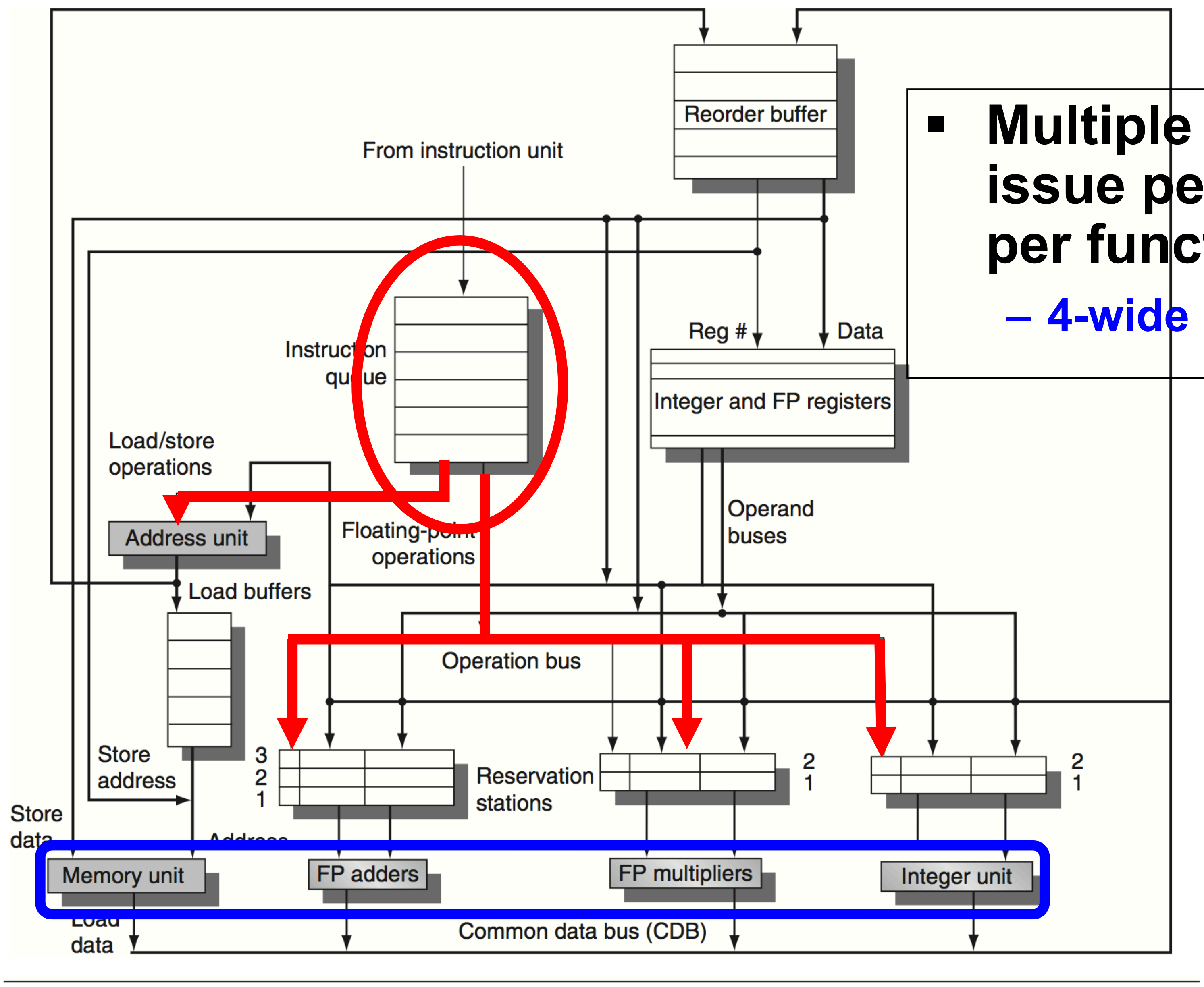
# 5-Stage In-order 2-Wide Pipeline



- what is involved in
  - fetching two instructions per cycle?
  - decoding two instructions per cycle?
  - executing two ALU operations per cycle?
  - accessing the data cache twice per cycle?
  - writing back two results per cycle?
- what about 4 or 8 instructions per cycle?

# Implementation using Temasulo's Approach

- Similar to Tomasulo with Speculation



- Multiple issue → one issue per clock cycle per functional unit
  - 4-wide

# Options and Challenges of Multiple Issue

- How to issue two instructions and keep in-order instruction issue for Tomasulo?
  - Assume 1 integer + 1 floating point
  - 1 Tomasulo control for integer, 1 for floating point

1. Issue two instrs pipelined in one cycle (half and half for each instr), so that issue remains in order → superpipelining
   - Hard to extend to 4 or more
2. Issue 2 instrs per cycle in parallel → true superscalar
   - Between FP and Integer operations: Only FP loads might cause dependency between integer and FP issue:
     - » Replace load reservation station with a load queue; operands must be read in the order they are fetched
     - » Load checks addresses in Store Queue to avoid RAW violation
     - » Store checks addresses in Load Queue to avoid WAR,WAW
     - » Called "decoupled architecture"
3. Mix of both
   - Superpipeling and superscalar

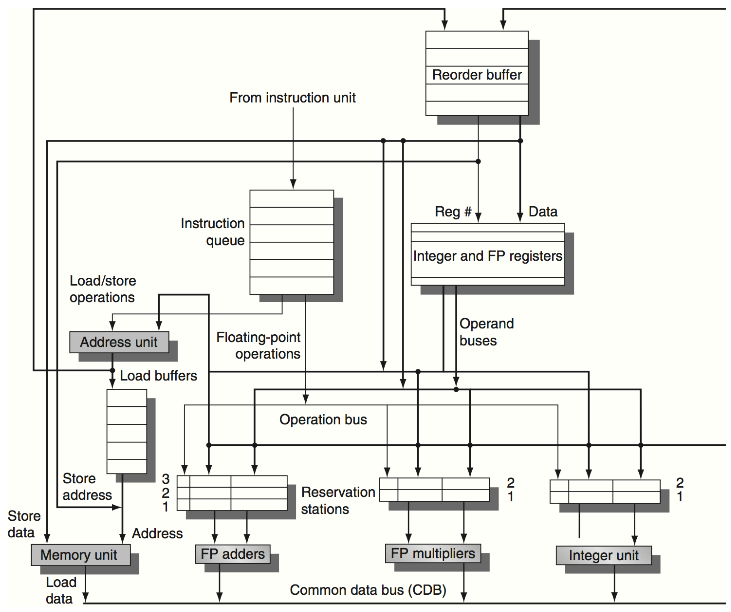
## Multiple Issue Challenges

- **While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:**
  - Exactly 50% FP operations
  - No hazards

- **If more instructions issue at same time, greater difficulty of decode and issue:**
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
  - Multiported rename logic: must be able to rename same register multiple times in one cycle!
  - Rename logic one of key complexities in the way of multiple issue!

# Multiple Issue

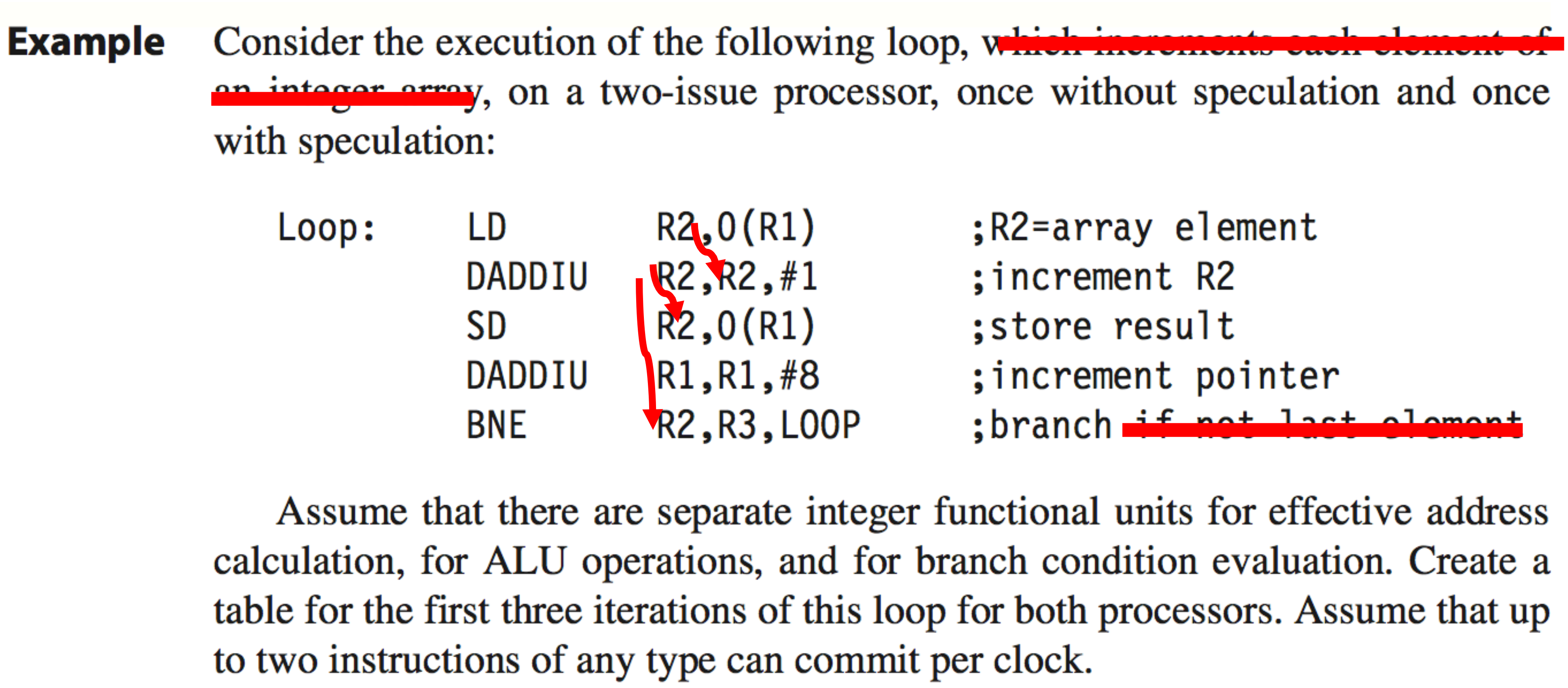
- Bundle multiple instrs in one issue unit
  - *N*-wide



1. Assign a reservation station and a reorder buffer for *every* instruction that *might* be issued in the next issue bundle
   - N entries in ROB
   - Ensure enough RS available for the bundle
   - If not enough RS/ROB, break the bundle
2. Analyze dependency in the issue bundle
3. Inter-dependency between instrs in a bundle
   - Update the reservation station table entries using the assigned ROB entries to link the dependency
     - » Register renaming happened

- In-order commit to make sure instrs commit in order
- Other techniques
  - Speculative multiple issue in Intel i7

# Example



**Example** Consider the execution of the following loop, ~~which increments each element of an integer array~~, on a two-issue processor, once without speculation and once with speculation:

```
Loop:    LD       R2,0(R1)        ;R2=array element
         DADDIU   R2,R2,#1        ;increment R2
         SD       R2,0(R1)        ;store result
         DADDIU   R1,R1,#8        ;increment pointer
         BNE      R2,R3,LOOP      ;branch ~~if not last element~~
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.

1

**BNE has RAW dependence on DADDIU**

## Without Speculation

```
Loop:   LD      R2,0(R1)     ;R2=array element
        DADDIU  R2,R2,#1     ;increment R2
        SD      R2,0(R1)     ;store result
        DADDIU  R1,R1,#8     ;increment pointer
        BNE     R2,R3,LOOP   ;branch ~~if not last~~
```

**LD can be issued but CANNOT be executed before BNE completes**

| Iteration number | Instructions | Issues at clock cycle number | Executes at clock cycle number | access at clock cycle number | Write CDB at clock cycle number | Comment |
|---|---|---|---|---|---|---|
| 1 | LD R2,0(R1) | 1 | 2 | 3 | 4 | First issue |
| 1 | DADDIU R2,R2,#1 | 1 | 5 | | 6 | Wait for LW |
| 1 | SD R2,0(R1) | 2 | 3 | 7 | | Wait for DADDIU |
| 1 | DADDIU R1,R1,#8 | 2 | 3 | | 4 | Execute directly |
| 1 | BNE R2,R3,LOOP | 3 | 7 | | **No Speculation** | Wait for DADDIU |
| 2 | LD R2,0(R1) | 4 | 8 | 9 | 10 | Wait for BNE |
| 2 | DADDIU R2,R2,#1 | 4 | 11 | | 12 | Wait for LW |
| 2 | SD R2,0(R1) | 5 | 9 | 13 | | Wait for DADDIU |
| 2 | DADDIU R1,R1,#8 | 5 | 8 | | 9 | Wait for BNE |
| 2 | BNE R2,R3,LOOP | 6 | 13 | | **No Speculation** | Wait for DADDIU |
| 3 | LD R2,0(R1) | 7 | 14 | 15 | 16 | Wait for BNE |
| 3 | DADDIU R2,R2,#1 | 7 | 17 | | 18 | Wait for LW |
| 3 | SD R2,0(R1) | 8 | 15 | 19 | | Wait for DADDIU |
| 3 | DADDIU R1,R1,#8 | 8 | 14 | | 15 | Wait for BNE |
| 3 | BNE R2,R3,LOOP | 9 | 19 | | | Wait for DADDIU |

**Figure 3.19 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation.** Note that the LD following the BNE cannot start execution earlier because it must wait until the branch

# With Speculation

```
oop:    LD        R2,0(R1)      ;R2=array element
        DADDIU    R2,R2,#1      ;increment R2
        SD        R2,0(R1)      ;store result
        DADDIU    R1,R1,#8      ;increment pointer
        BNE       R2,R3,LOOP    ;branch ~~if not last~~
```

1 2 3

LD can be speculatively executed before BNE completes

| Iteration number | Instructions | | Issues at clock number | Execute at clock number | at clock number | clock number | at clock number | Comment |
|---|---|---|---|---|---|---|---|---|
| 1 | LD | R2,0(R1) | 1 | 2 | 3 | 4 | 5 | First issue |
| 1 | DADDIU | R2,R2,#1 | 1 | 5 | | 6 | 7 | Wait for LW |
| 1 | SD | R2,0(R1) | 2 | 3 | | | 7 | Wait for DADDIU |
| 1 | DADDIU | R1,R1,#8 | 2 | 3 | | 4 | 8 | Commit in order |
| 1 | BNE | R2,R3,LOOP | 3 | 7 | | | 8 | Wait for DADDIU |
| 2 | LD | R2,0(R1) | 4 | 5 | 6 | 7 | 9 | No execute delay |
| 2 | DADDIU | R2,R2,#1 | 4 | 8 | | 9 | 10 | Wait for LW |
| 2 | SD | R2,0(R1) | 5 | 6 | | | 10 | Wait for DADDIU |
| 2 | DADDIU | R1,R1,#8 | 5 | 6 | | 7 | 11 | Commit in order |
| 2 | BNE | R2,R3,LOOP | 6 | 10 | | | 11 | Wait for DADDIU |
| 3 | LD | R2,0(R1) | 7 | 8 | 9 | 10 | 12 | Earliest possible |
| 3 | DADDIU | R2,R2,#1 | 7 | 11 | | 12 | 13 | Wait for LW |
| 3 | SD | R2,0(R1) | 8 | 9 | | | 13 | Wait for DADDIU |
| 3 | DADDIU | R1,R1,#8 | 8 | 9 | | 10 | 14 | Executes earlier |
| 3 | BNE | R2,R3,LOOP | 9 | 13 | | | 14 | Wait for DADDIU |

With Speculation

Gain only 1 clock per iteration

**Figure 3.20 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation.** Note that the LD following the BNE can start execution early because it is speculative.

## Putting It All Together: The Intel Core i7 6700 and ARM Cortex-A53

Textbook CAQA 3.12

## Reality and References

- Modern processors uses the advanced technologies we talked about in this class and some others that are not covered
  - Principles are the same mostly

- Historically and more depth
  - Lots of ideas have been evaluated and developed
  - Appendix L.5 for history and references
  - VLIW/EPIC and software pipelining: Appendix H
- More and Latest Info (Conference)
  - MICRO: Annual IEEE/ACM International Symposium on Microarchitecture
    - » https://www.microarch.org
  - IEEE Symposium on High Performance Computer Architecture (HPCA)
    - » http://hpca2017.org/
  - International Symposium on Computer Architecture (ISCA)
  - ACM International Conference on Architectural Support for Programming Languages and Operating Systems
    - » http://www.ece.cmu.edu/calcm/asplos2016
  - SIGARCH – The ACM Special Interest Group on Computer Architecture
    - » https://www.sigarch.org/

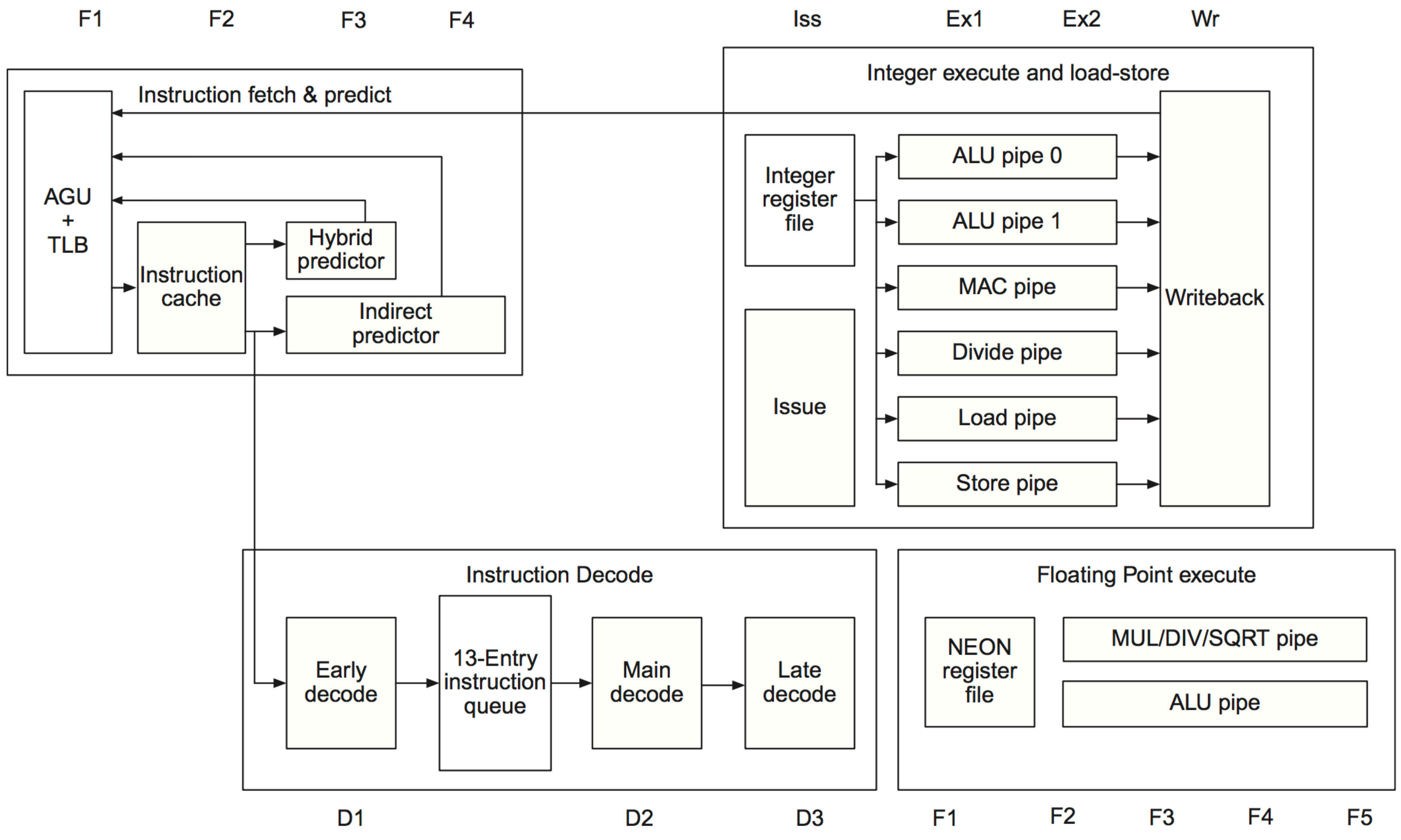
# Intel Core i7 6700 and ARM Cortex-A53

- **ARM Cortex-A53 core**
  - Used as the basis for several tablets and cell phones

- **Intel Core i7 6700**
  - a high-end, dynamically scheduled, speculative processor intended for high-end desktops and server applications.

# ARM Cortex-A53

- Used as the basis for several tablets and cell phones
  - Dual-issue, statically scheduled superscalar with dynamic issue detection → 0.5 CPI ideally



# ARM Cortex-A53 Missprediction Rate

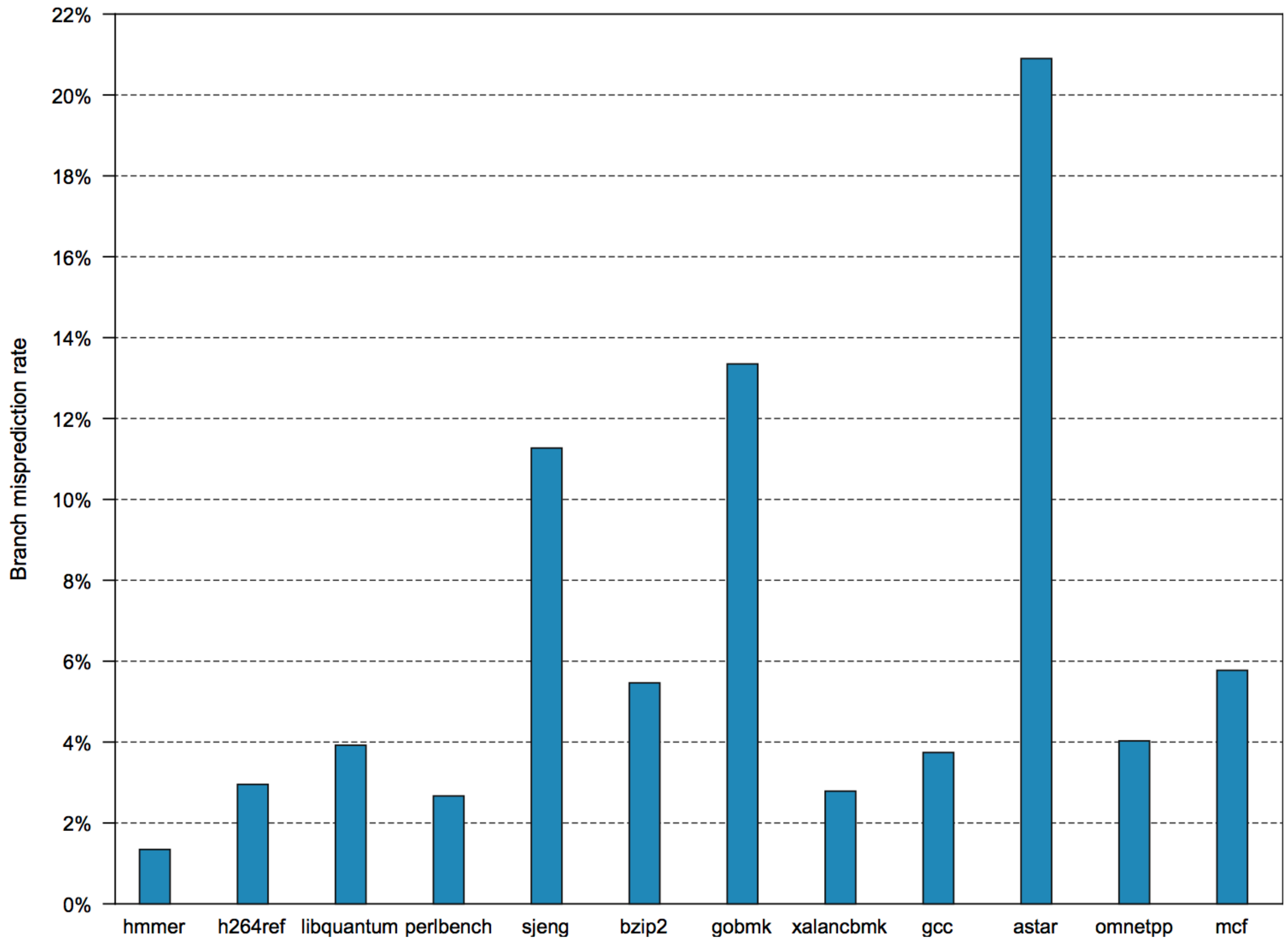


**Figure 3.35** **Misprediction rate of the A53 branch predictor for SPECint2006.**

# Wasted Word Due to Misprediction on A53

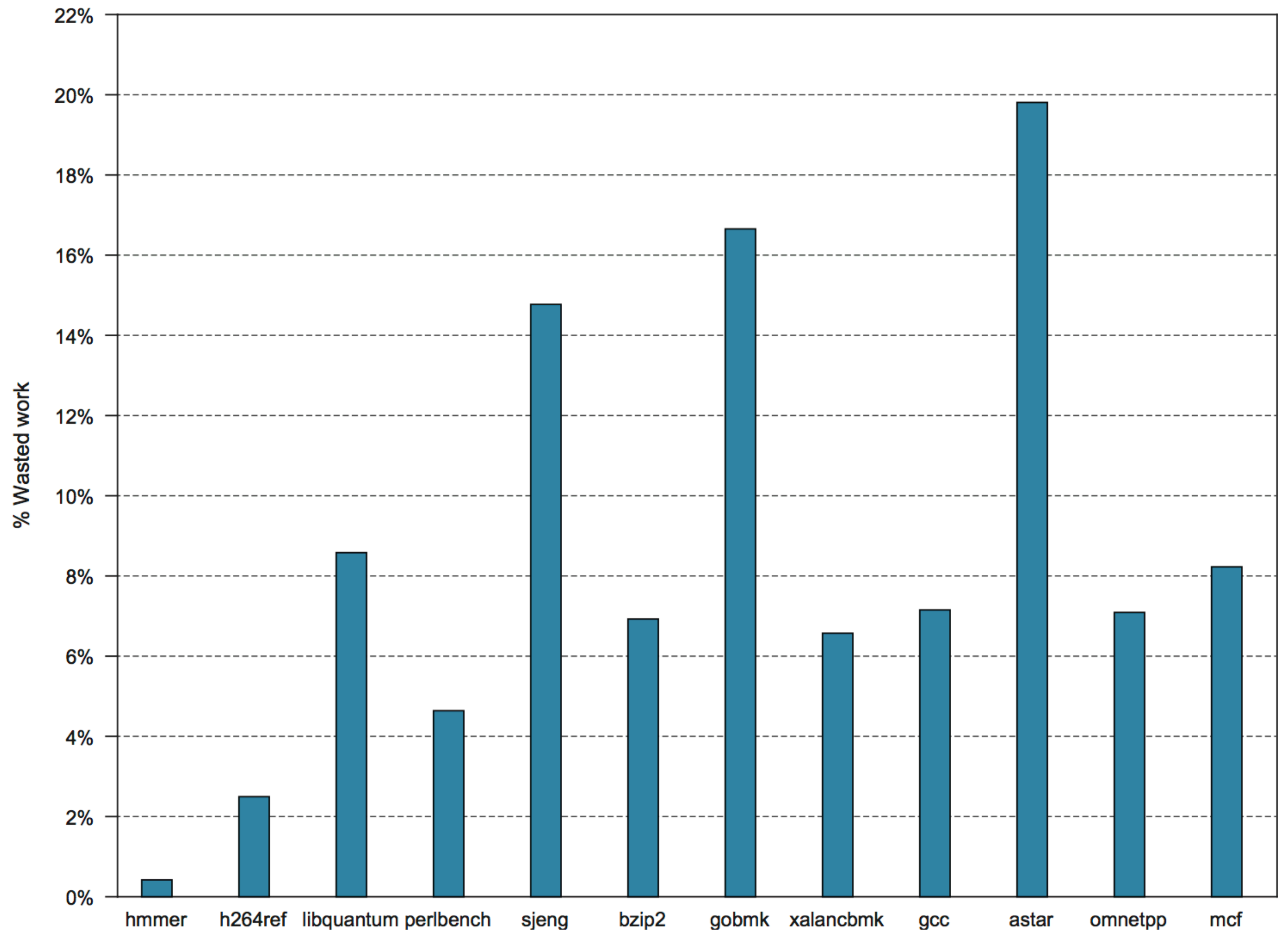


**Figure 3.36 Wasted work due to branch misprediction on the A53.** Because the A53 is an in-order machine, the amount of wasted work depends on a variety of factors, including data dependences and cache misses, both of which will cause a stall.

# Estimated Composition of ARM A53 CPI

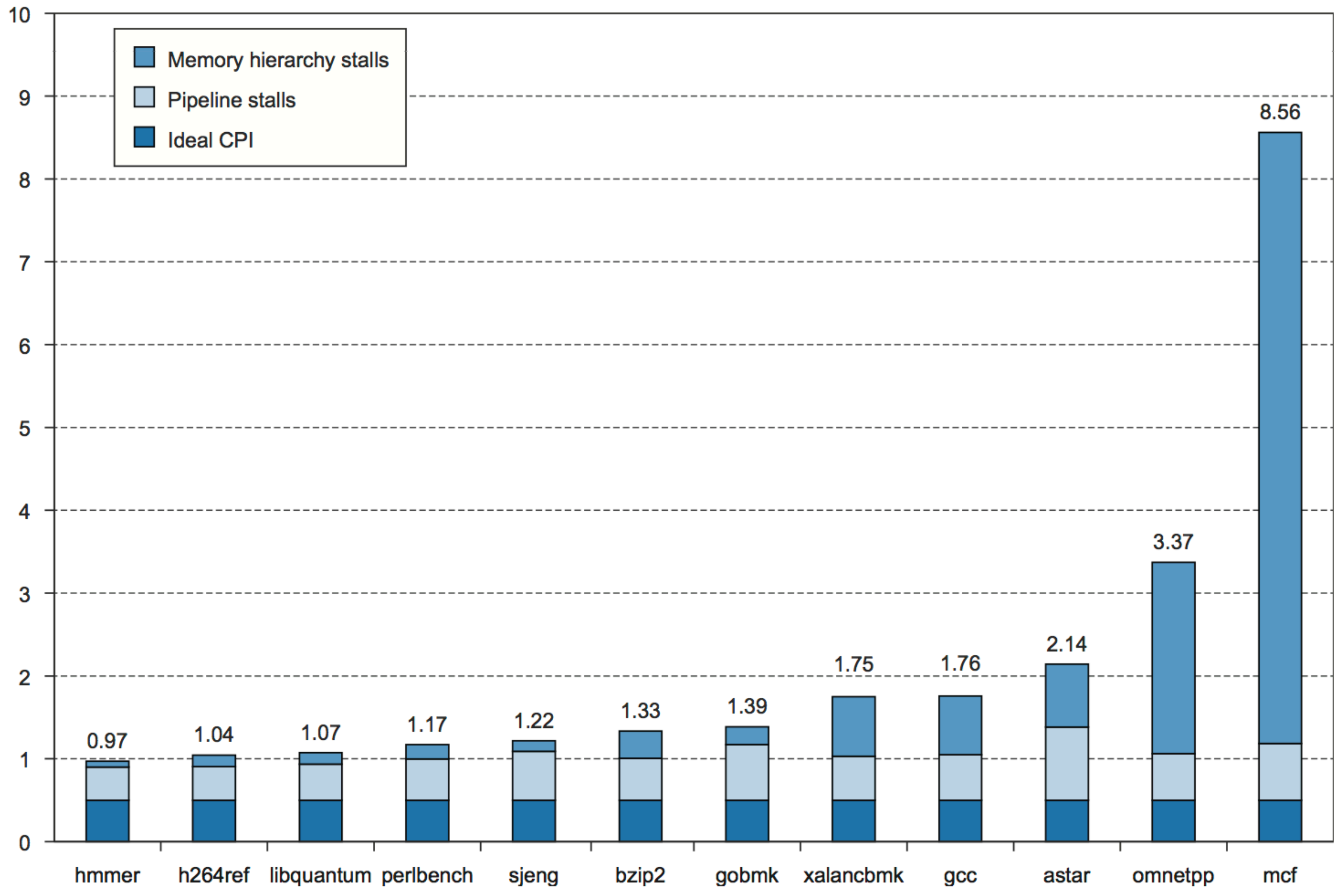
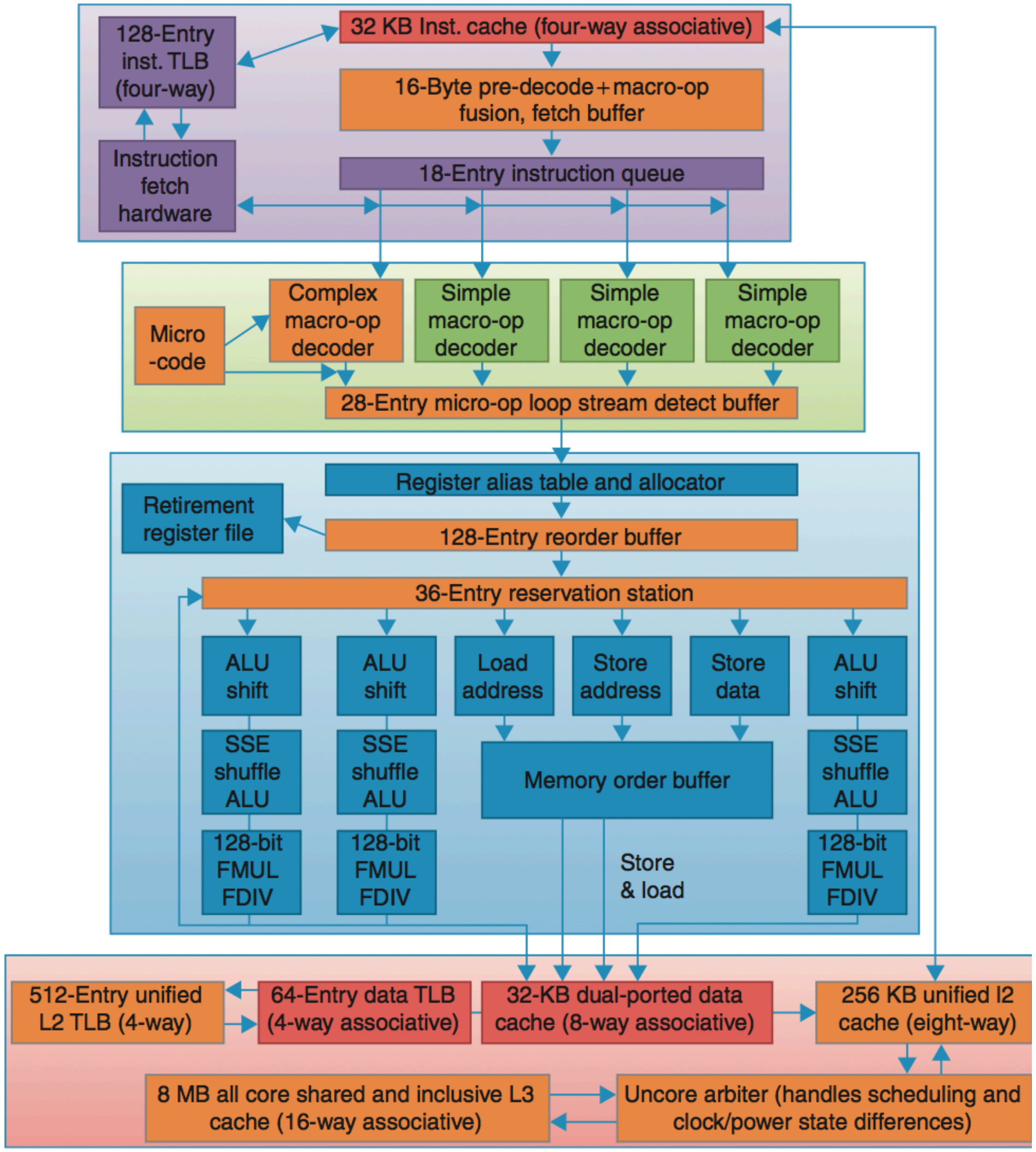


**Figure 3.37 The estimated composition of the CPI on the ARM A53 shows that pipeline stalls are significant but are outweighed by cache misses in the poorest performing programs.** This estimate is obtained by using the L1 and L2 miss rates and penalties to compute the L1 and L2 generated stalls per instruction. These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards.

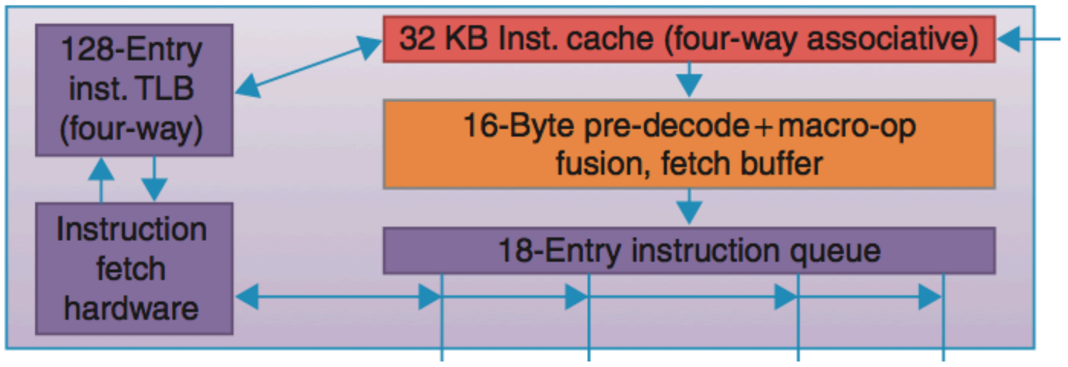
# Intel Core i7

- Aggressive out-of-order speculative
- 14 stages pipeline,
- Branch mispredictions costing 17 cycles.
- 48 load and 32 store buffers.
- Six independent functional units
  – 6-wide superscalar



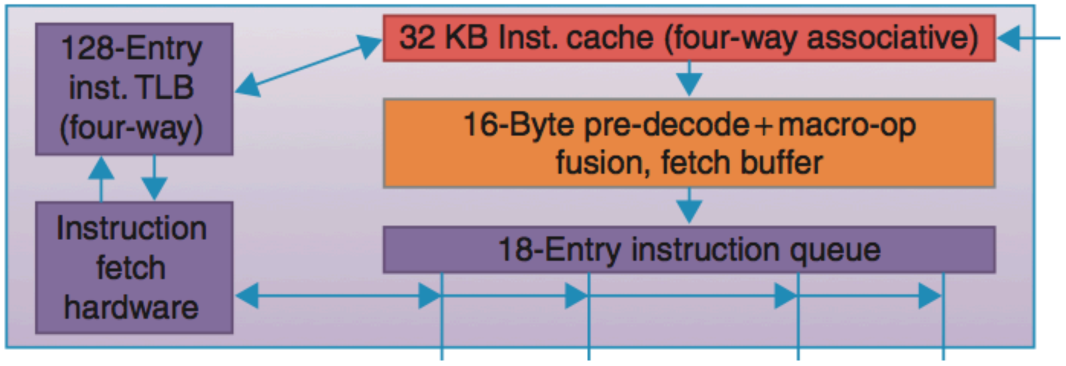
128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode + macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
128-Entry reorder buffer
36-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
512-Entry unified L2 TLB (4-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

## Core i7 Pipeline: IF


128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue

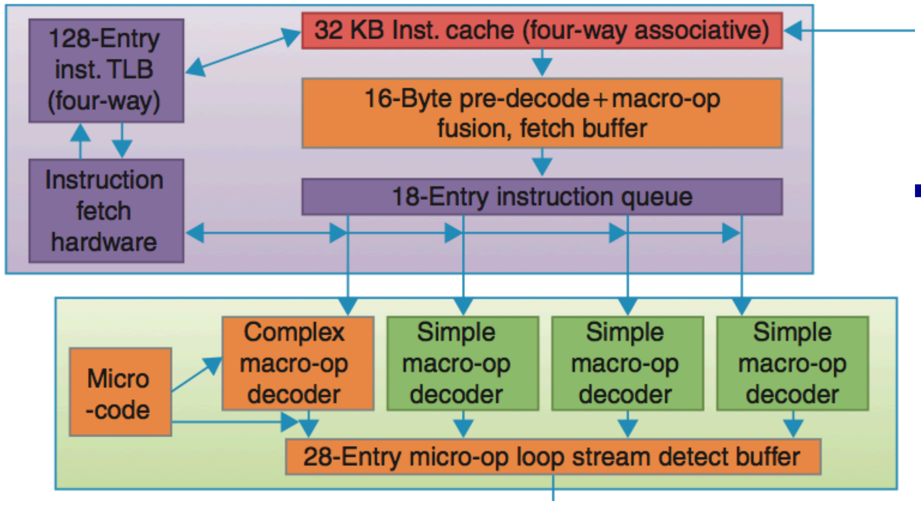
- **Instruction fetch – Fetch 16 bytes from the I cache**
  - **A multilevel branch target buffer to achieve a balance between speed and prediction accuracy.**
  - **A return address stack to speed up function return.**
  - **Mispredictions cause a penalty of about 15 cycles.**

# Core i7 Pipeline: Predecode


128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue

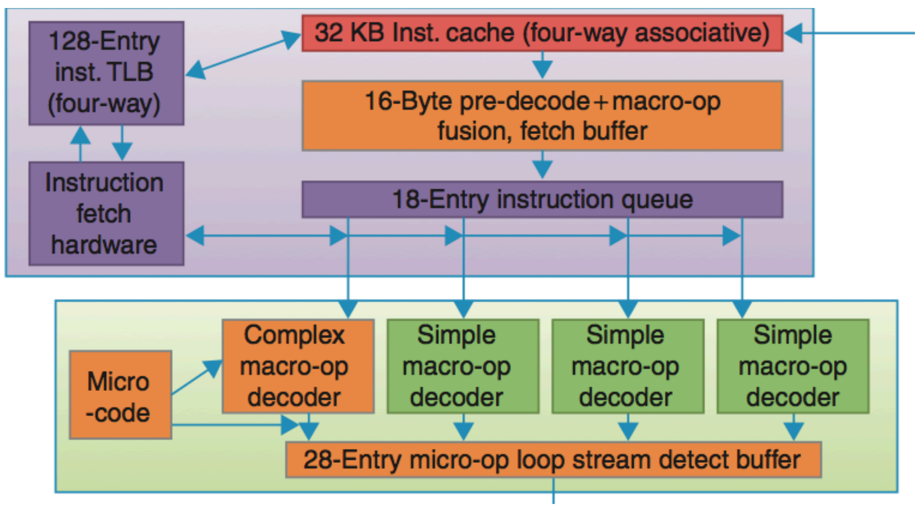
- **Predecode –16 bytes instr in the predecode I buffer**
  - ***Macro-op fusion:*** **Fuse instr combinations such as compare followed by a branch into a single operation.**
  - **Instr break down: breaks the 16 bytes into individual x86 instructions.**
    - » **nontrivial since the length of an x86 instruction can be from 1 to 17 bytes and the predecoder must look through a number of bytes before it knows the instruction length.**
  - **Individual x86 instructions (including some fused instructions) are placed into the 18-entry instruction queue.**

# Core i7 Pipeline: Micro-op decode


128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer

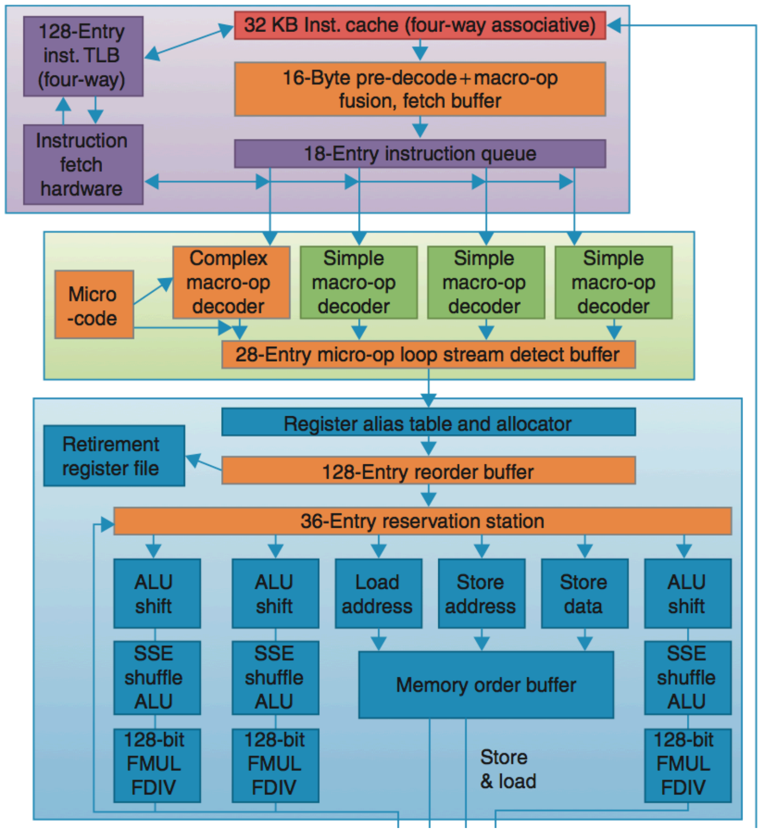
- Micro-op decode – Translate Individual x86 instructions into micro-ops.
  - Micro-ops are simple MIPS-like instructions that can be executed directly by the pipeline (RISC style)
    - » introduced in the Pentium Pro in 1997 and has been used since.
  - Three simple micro-op decoders handle x86 instructions that translate directly into one micro-op.
  - One complex micro-op decoder produce the micro-op sequence of complex x86 instr;
    - » produce up to four micro-ops every cycle
  - The micro-ops are placed according to the order of the x86 instructions in the 28- entry micro-op buffer.

# Core i7 Pipeline: loop stream detection and microfusion


128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer

- *loop stream detection* and *microfusion by* the micro-op buffer preforms
  - If there is a sequence of instructions (less than 28 instrs or 256 bytes in length) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-ops from the buffer
    - » eliminating the need for the instruction fetch and instruction decode stages to be activated.
  - Microfusion combines instr pairs such as load/ALU operation and ALU operation/store and issues them to a single reservation station, thus increasing the usage of the buffer.
    - » Study comparing the microfusion and macrofusion by Bird et al. [2007] discovered that microfusion had little impact on performance, while macrofusion appears to have a modest positive impact on integer performance and little impact on FP.

# Core i7 Pipeline: Issue

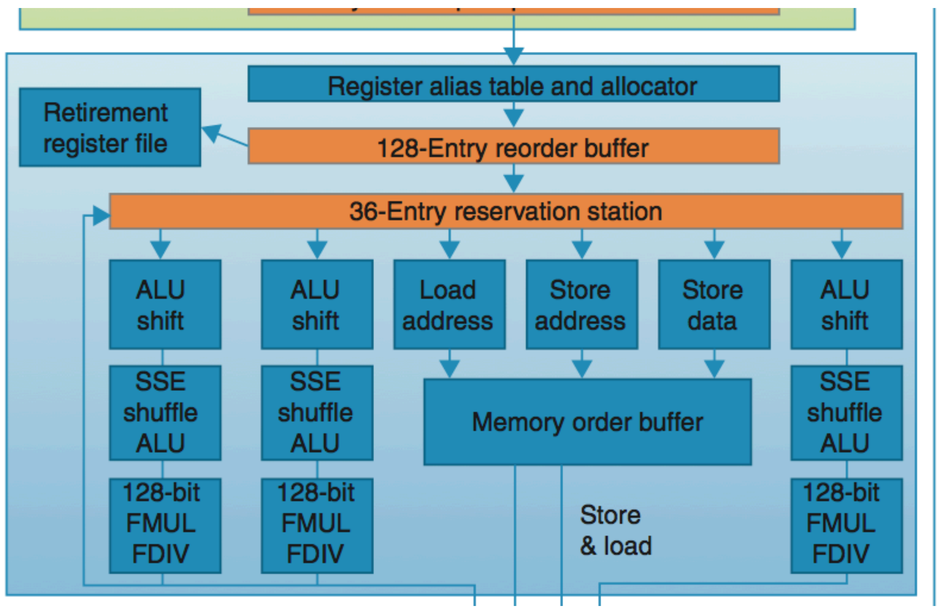


- **Basic instruction issue**
  - Looking up the register location in the register tables
  - renaming the registers
  - allocating a reorder buffer entry
  - fetching any results from the

  registers or reorder buffer before sendi
  reservation stations.
- **36-entry centralized reservation station shared by six functional units**

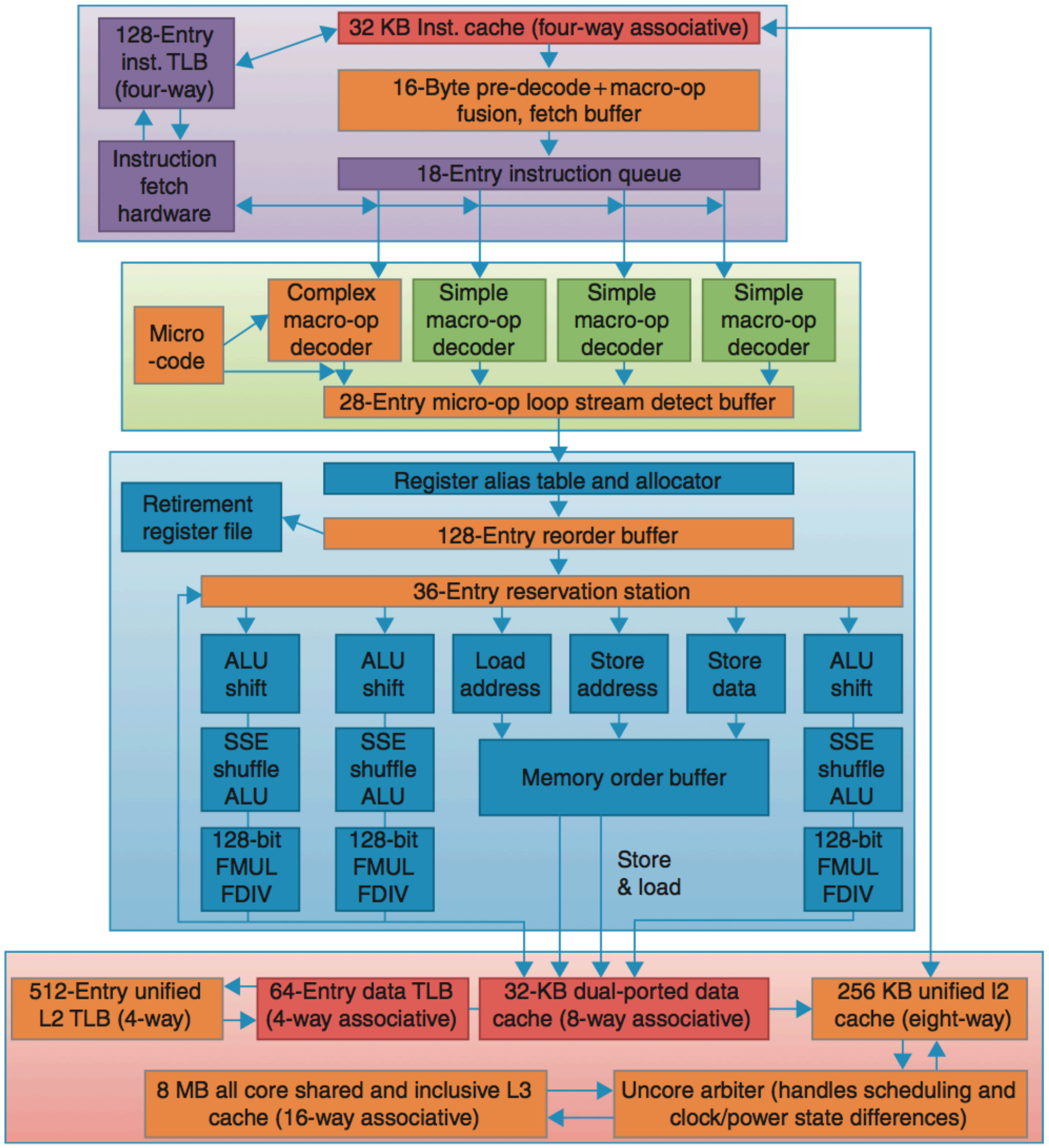
  Up to six micro-ops may be dispatched to the functional units every clock cycle.

## Core i7 Pipeline: EXE and Retirement



- Micro-ops are executed by the individual function units
  - results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state. The entry corresponding to the instruction in the reorder buffer is marked as complete.
- Retirement
  - When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.

# Intel Core i7: 970 (Nehalem, 2008) vs 6700 (Skylake, 2015)


128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
128-Entry reorder buffer
36-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
512-Entry unified L2 TLB (4-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

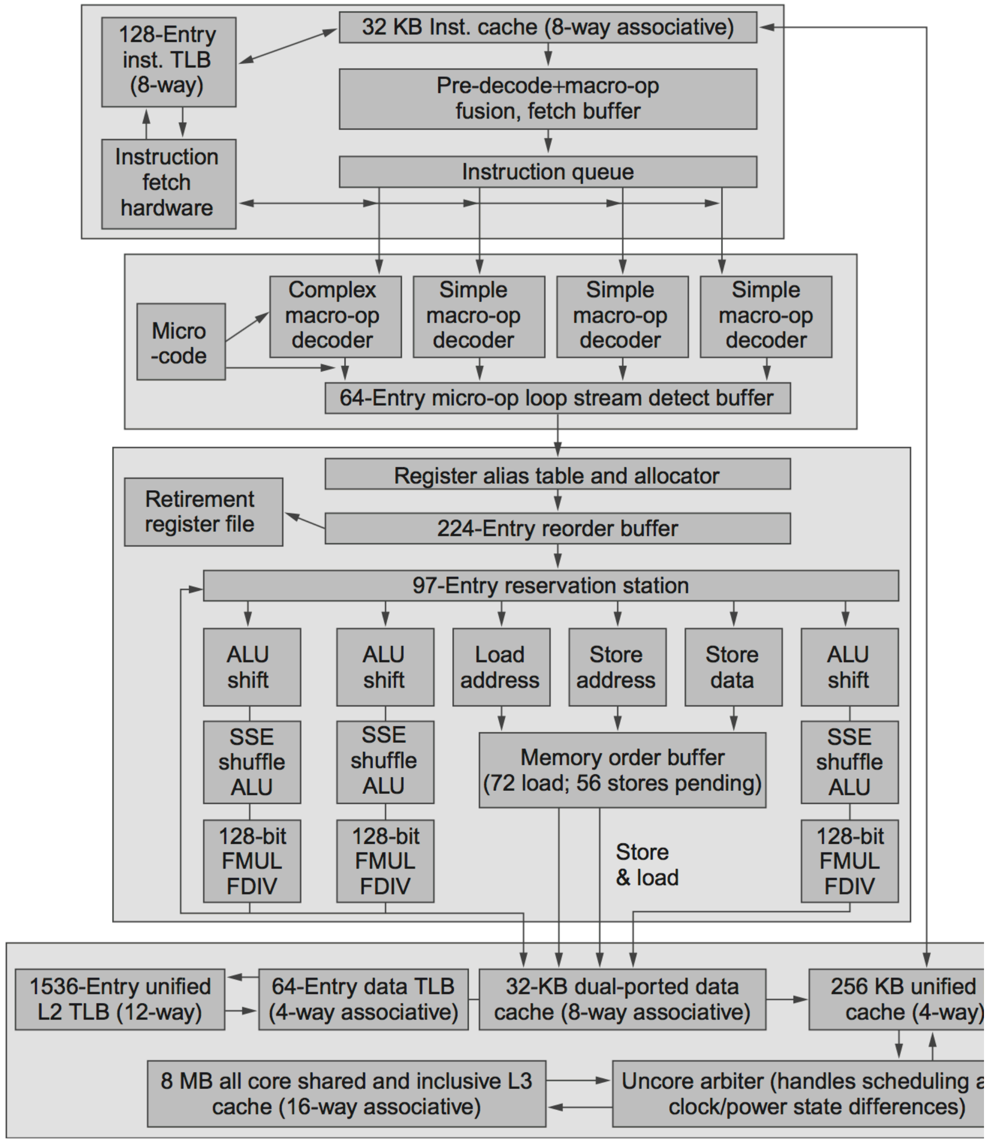
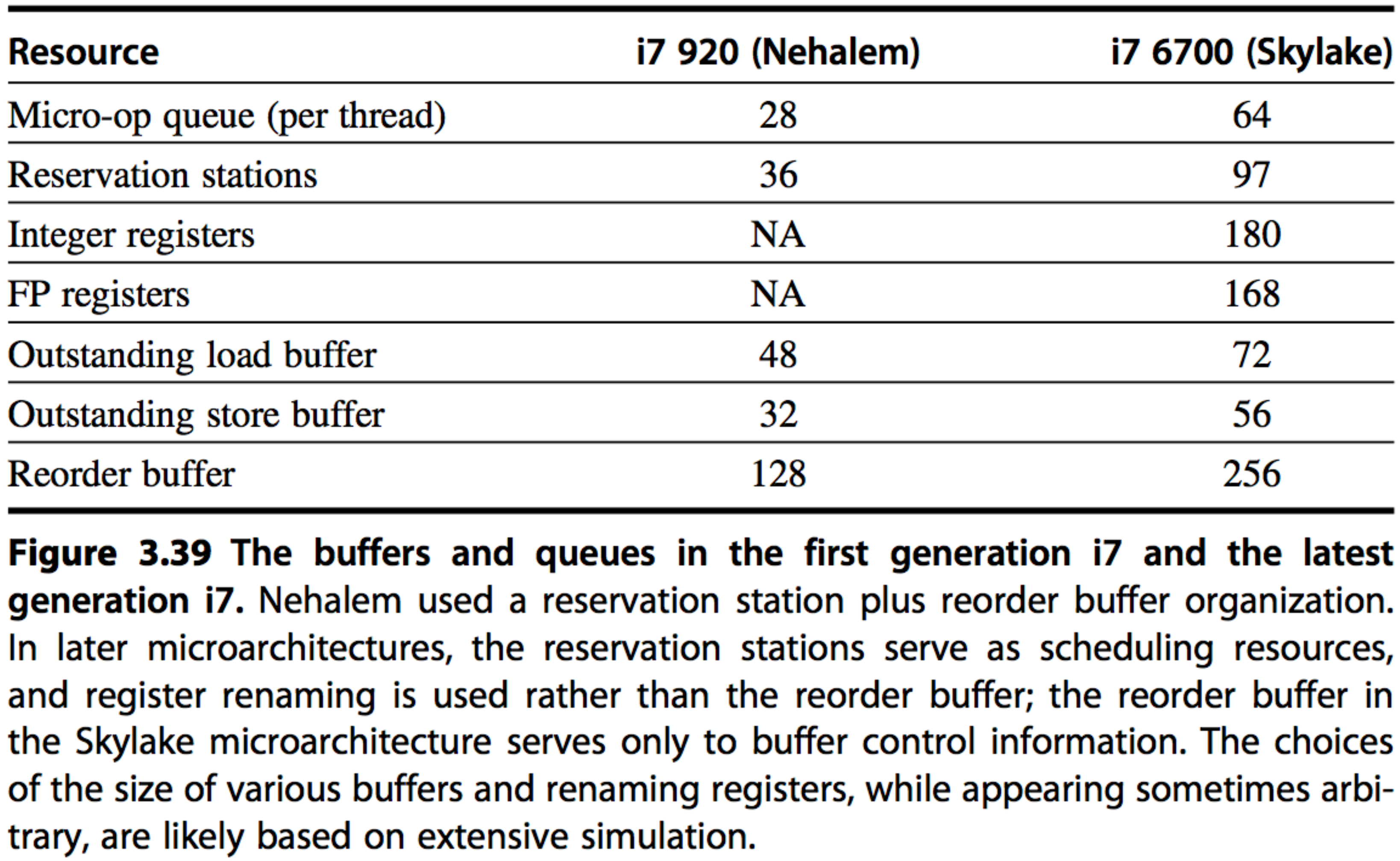

128-Entry inst. TLB (8-way)
32 KB Inst. cache (8-way associative)
Pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
Instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
64-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
224-Entry reorder buffer
97-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer (72 load; 56 stores pending)
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
1536-Entry unified L2 TLB (12-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified cache (4-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling a
clock/power state differences)

**Figure 3.38 The Intel Core i7 pipeline structure shown with the memory system components.** The depth is 14 stages, with branch mispredictions typically costing 17 cycles, with the extra few cycles like time to reset the branch predictor. The six independent functional units can each begin execution of a re in the same cycle. Up to four micro-ops can be processed in the register renaming table.

# Intel Core i7: 970 (Nehalem, 2008) vs 6700 (Skylake, 2015)



| Resource | i7 920 (Nehalem) | i7 6700 (Skylake) |
|---|---|---|
| Micro-op queue (per thread) | 28 | 64 |
| Reservation stations | 36 | 97 |
| Integer registers | NA | 180 |
| FP registers | NA | 168 |
| Outstanding load buffer | 48 | 72 |
| Outstanding store buffer | 32 | 56 |
| Reorder buffer | 128 | 256 |

**Figure 3.39 The buffers and queues in the first generation i7 and the latest generation i7.** Nehalem used a reservation station plus reorder buffer organization. In later microarchitectures, the reservation stations serve as scheduling resources, and register renaming is used rather than the reorder buffer; the reorder buffer in the Skylake microarchitecture serves only to buffer control information. The choices of the size of various buffers and renaming registers, while appearing sometimes arbitrary, are likely based on extensive simulation.

# Core i7 Performance

- **The integer CPI values range from 0.44 to 2.66 with a standard deviation of 0.77**
- **The FP CPU is from 0.62 to 1.38 with a standard deviation of 0.25.**
- **Cache behavior is major contribution to the stall CPI**

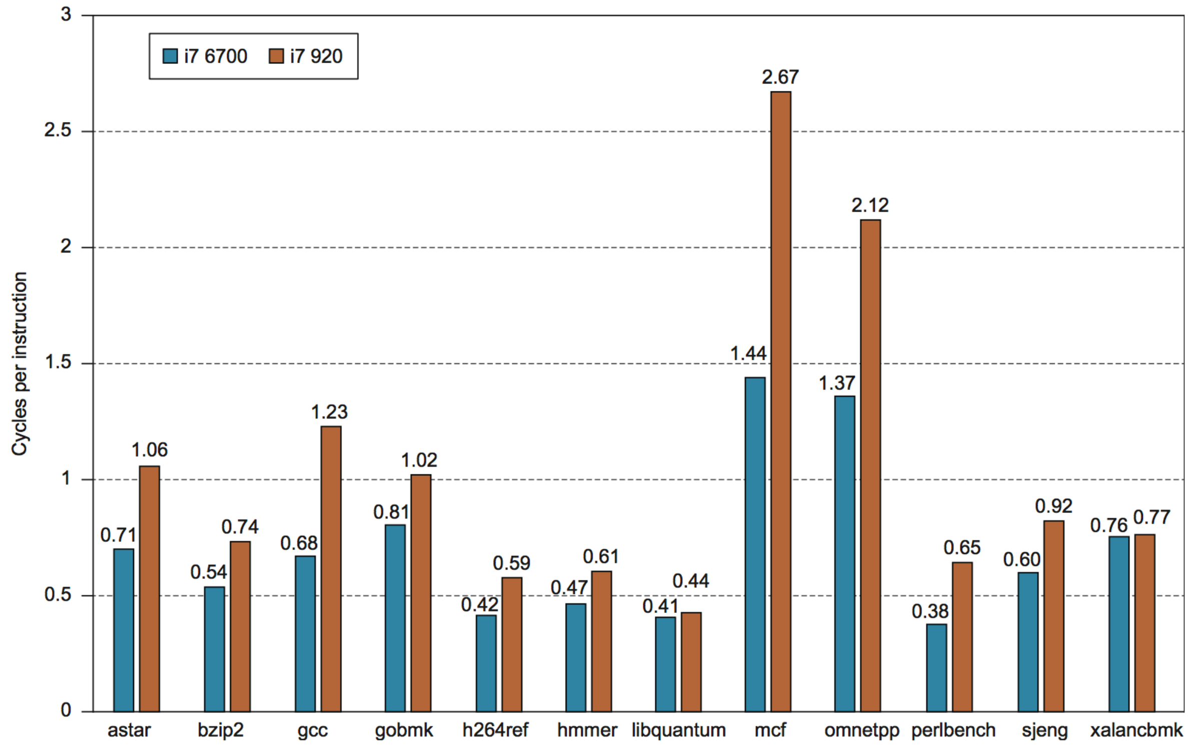


**Figure 3.40 The CPI for the SPECCPUint2006 benchmarks on the i7 6700 and the i7 920.** The data in this section were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University.

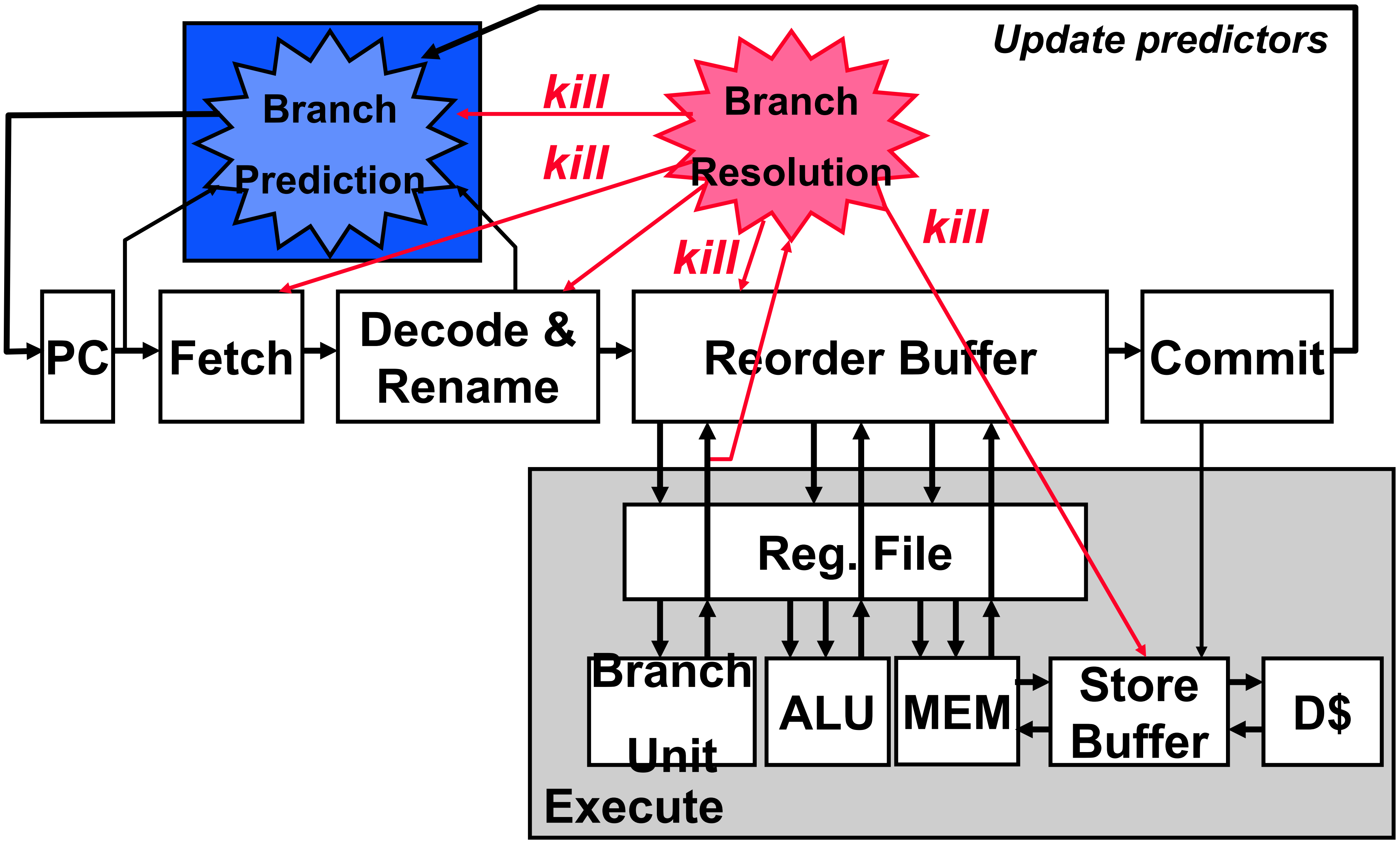
## Class Lectures End Here.

# Advanced Techniques for Instruction Delivery and Speculation

## Textbook CAQA 3.9

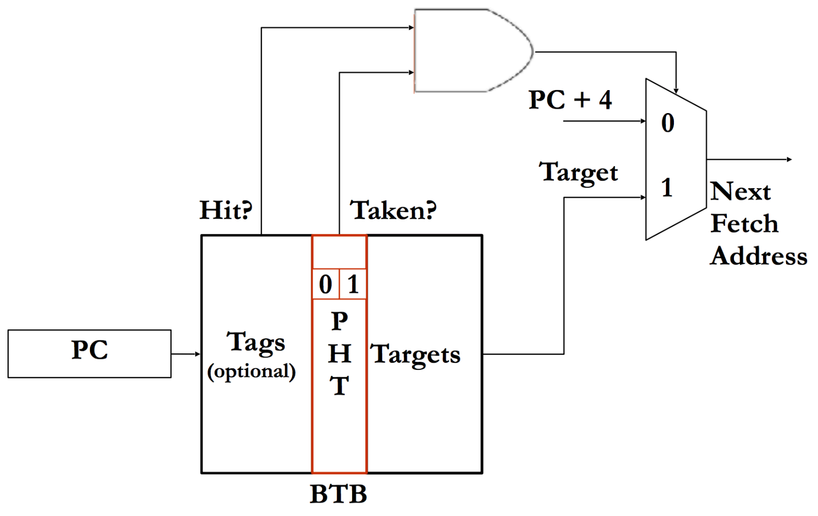
1. Improving Branch Prediction
2. Explicit Register Renaming
3. Others that are important but not covered: Load/store speculation, value predication, correlate branch prediction, tournament predictor, trace cache
4. Put all together on ARM Cortex-A53 and Intel Core i7 6700

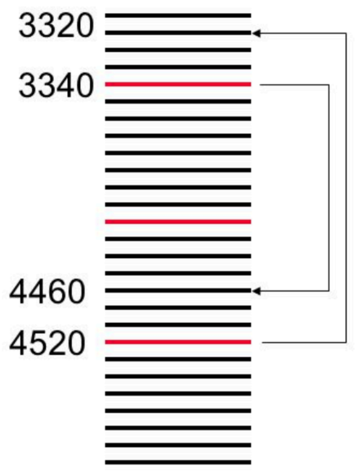
# Speculation: Prediction + Mis-prediction Recovery



# Branch Target Buffer for Branch Prediction

- Hardware support
  - Branch history tables (Taken or Not)
  - Branch target buffers, etc. (Target address)
- Branch target buffer
  - Cache for branch target

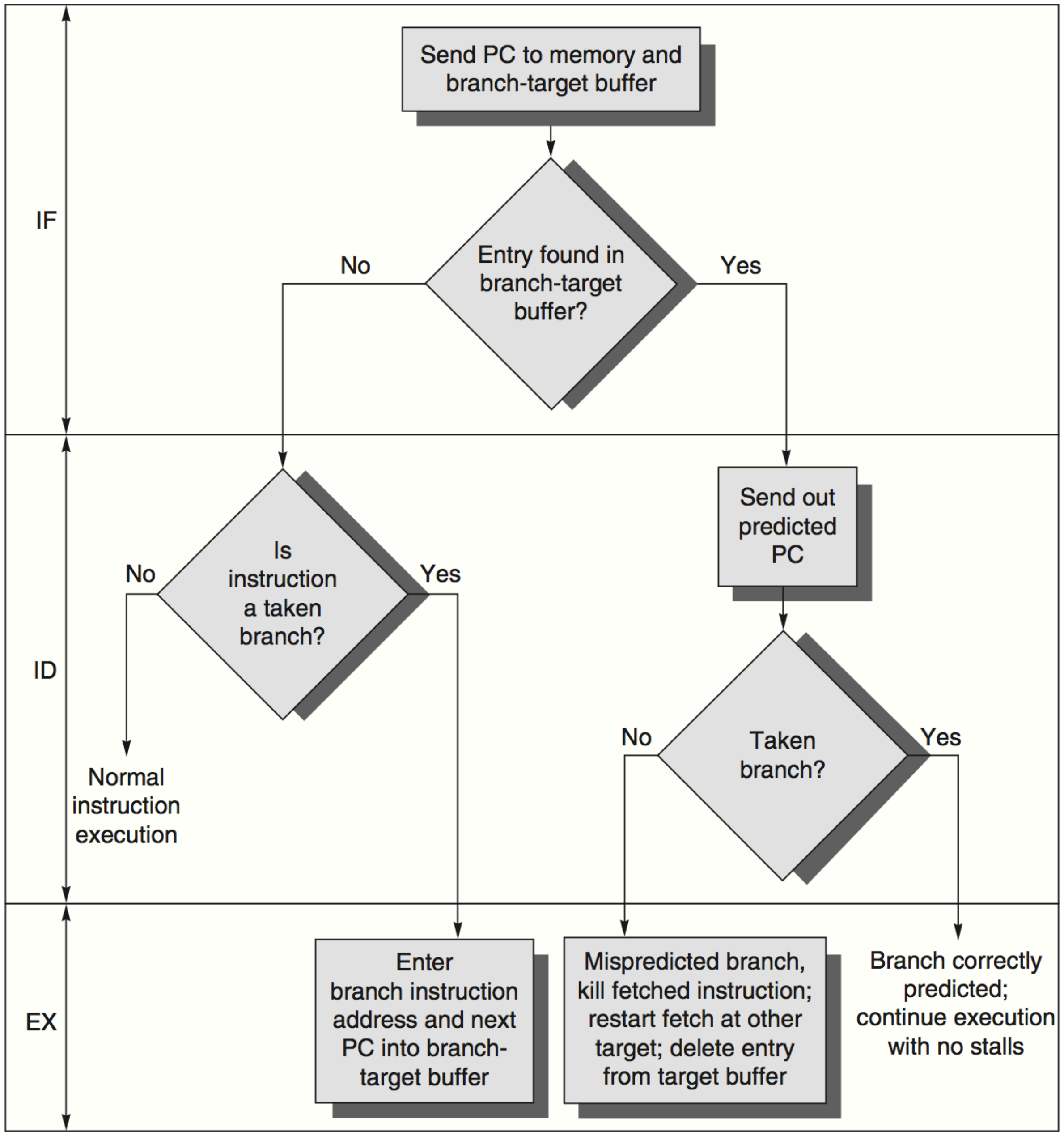




| PC | Target PC | Prediction |
|---|---|---|
| | | |
| 3340 | 4460 | 1(T) |
| | | |
| 4520 | 3320 | 1(T) |
| | | |
| | | |
| | | |
| | | |

# Branch With a Target Buffer

- **Steps**

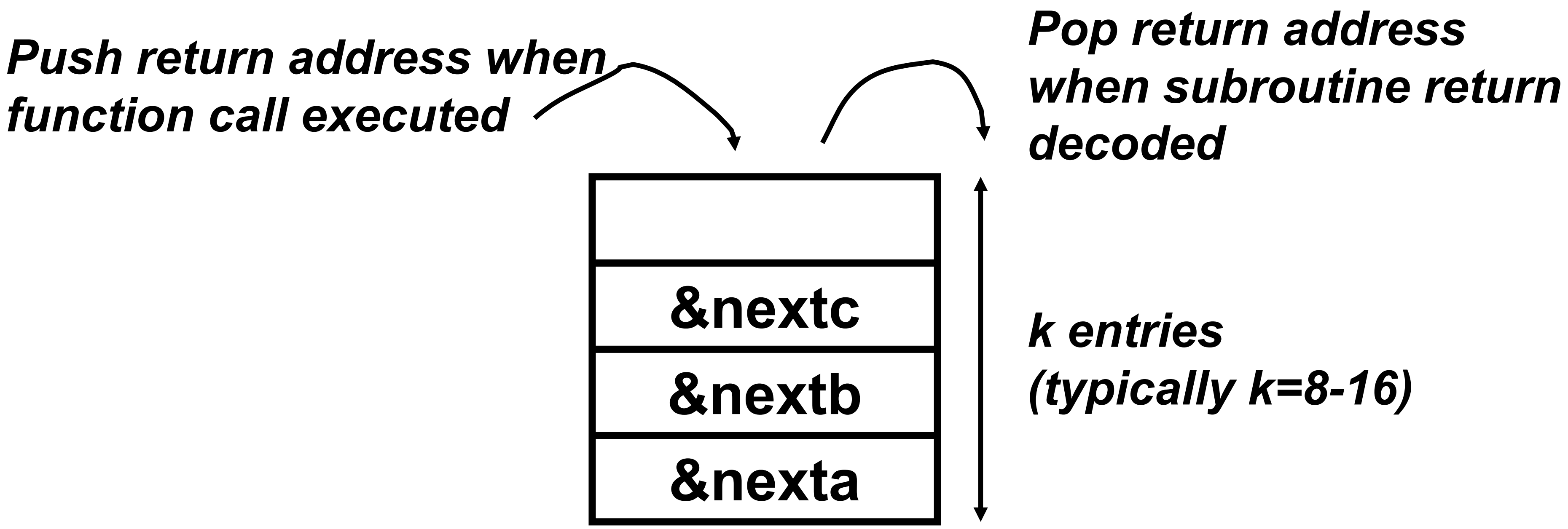


IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

## Subroutine Return Stack

**Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.**

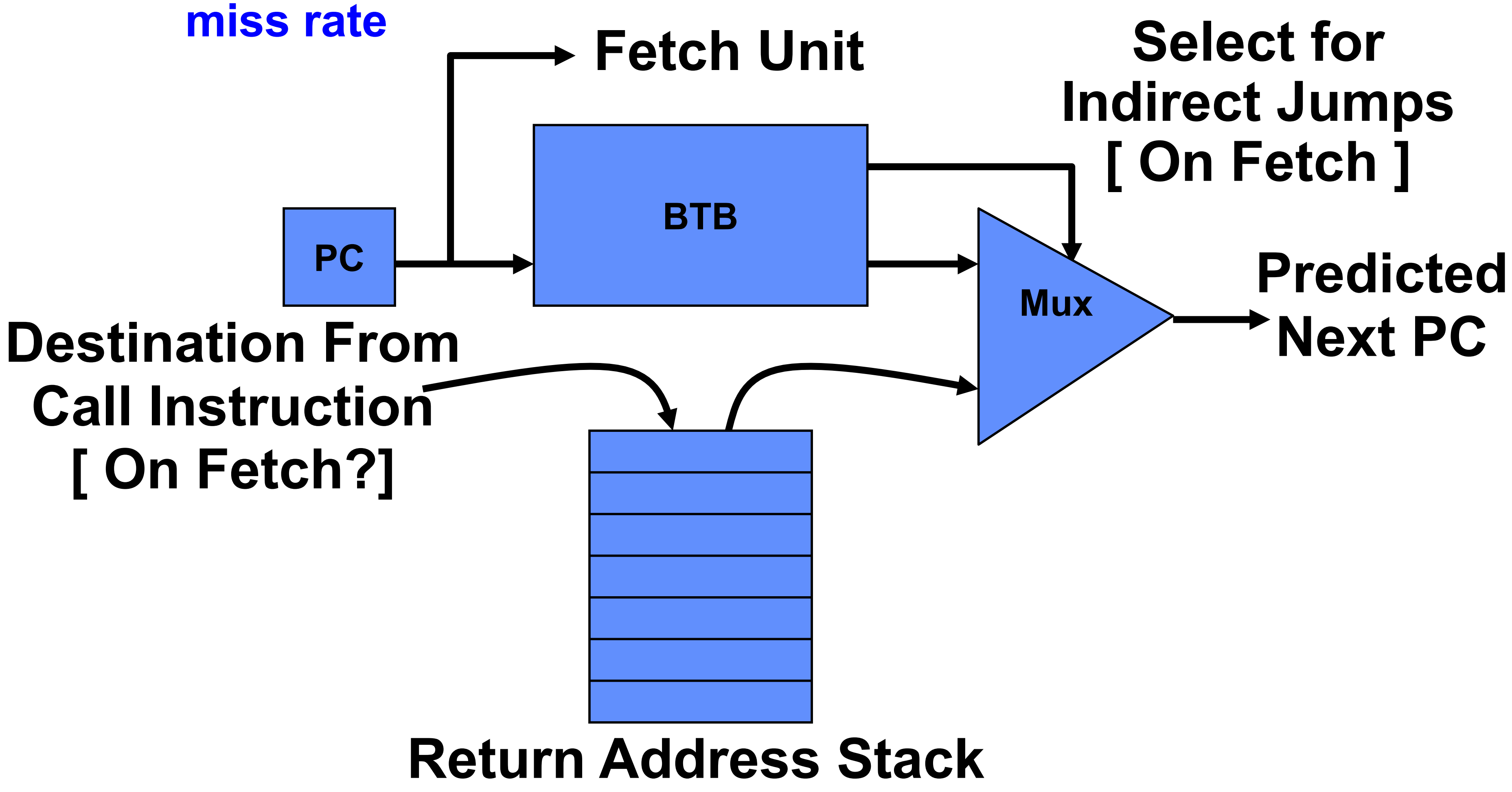
```
fa() { fb(); nexta: }

fb() { fc(); nextb: }

fc() { fd(); nextc: }
```



Push return address when function call executed
Pop return address when subroutine return decoded
&nextc
&nextb
&nexta
k entries
(typically k=8-16)

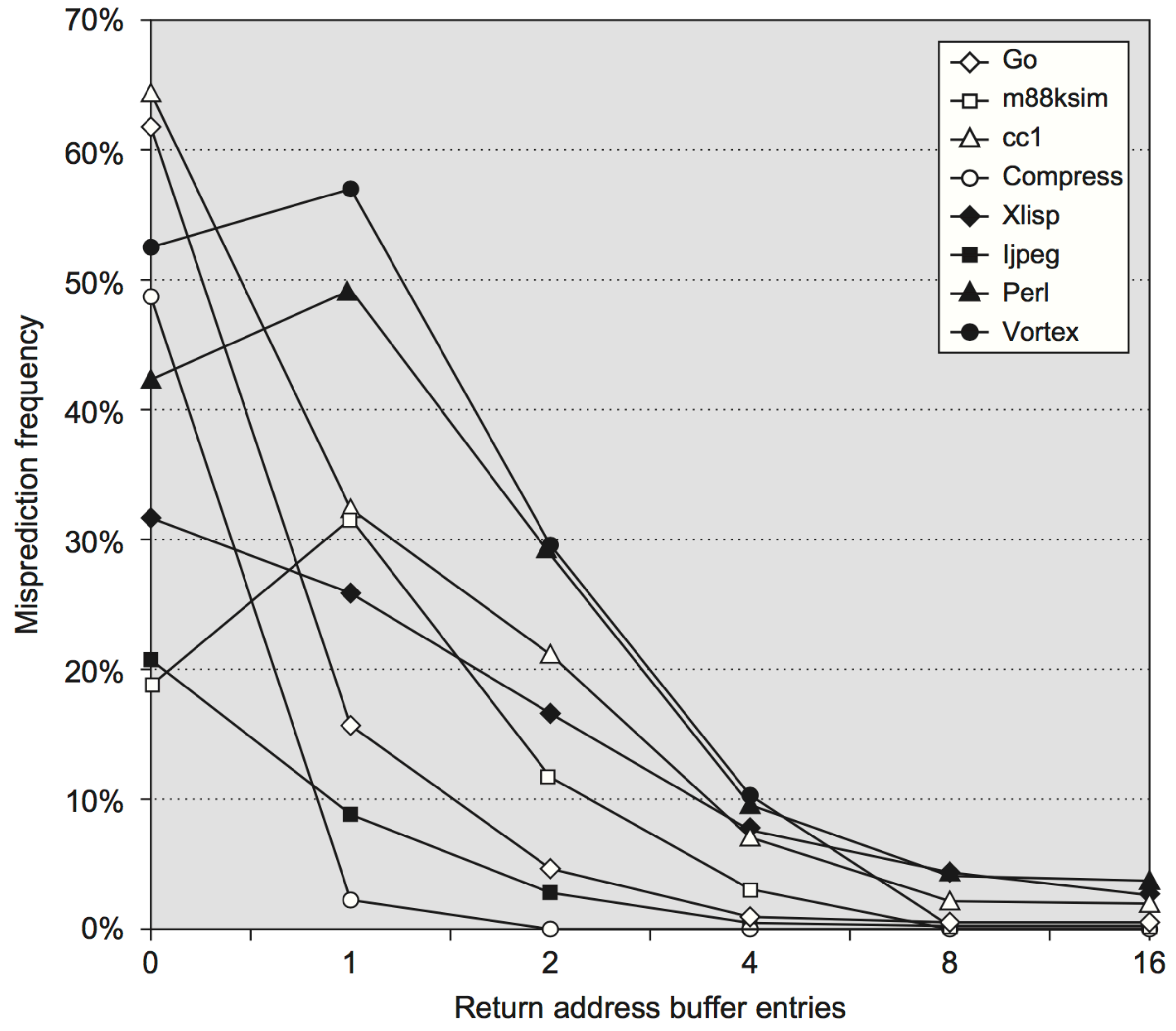
# Special Case Return Addresses

- Register Indirect branch hard to predict address
  - SPEC89 85% such branches for procedure return
  - Since stack discipline for procedures, save return address in small buffer that acts like a stack: 8 to 16 entries has small miss rate

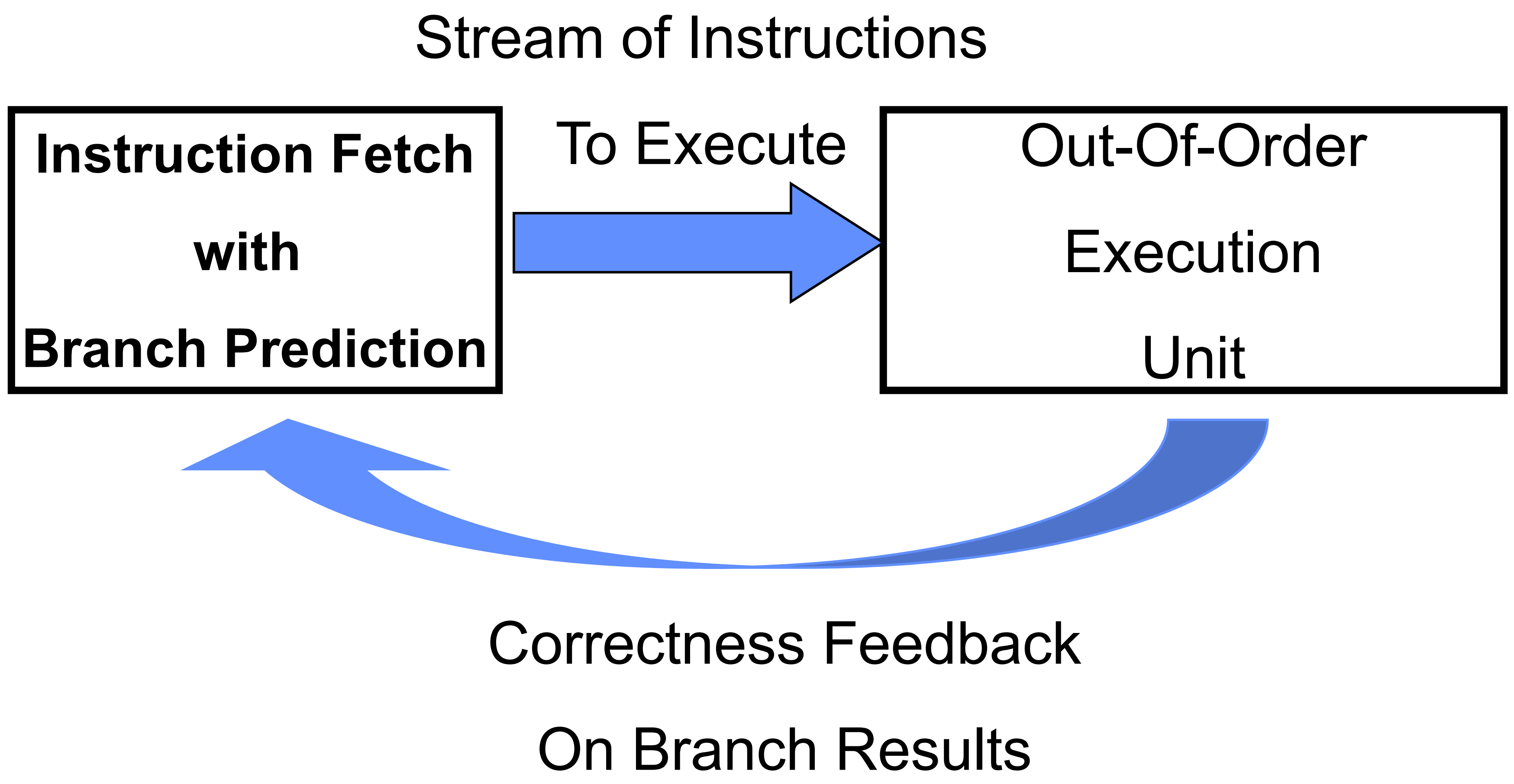

Fetch Unit
Select for Indirect Jumps [ On Fetch ]
BTB
PC
Mux
Predicted Next PC
Destination From Call Instruction [ On Fetch?]
Return Address Stack

# Performance: Return Address Predictor

- **Cache most recent return addresses:**
  - **Call ➔Push a return address on stack**
  - **Return ➔ Pop an address off stack & predict as new PC**

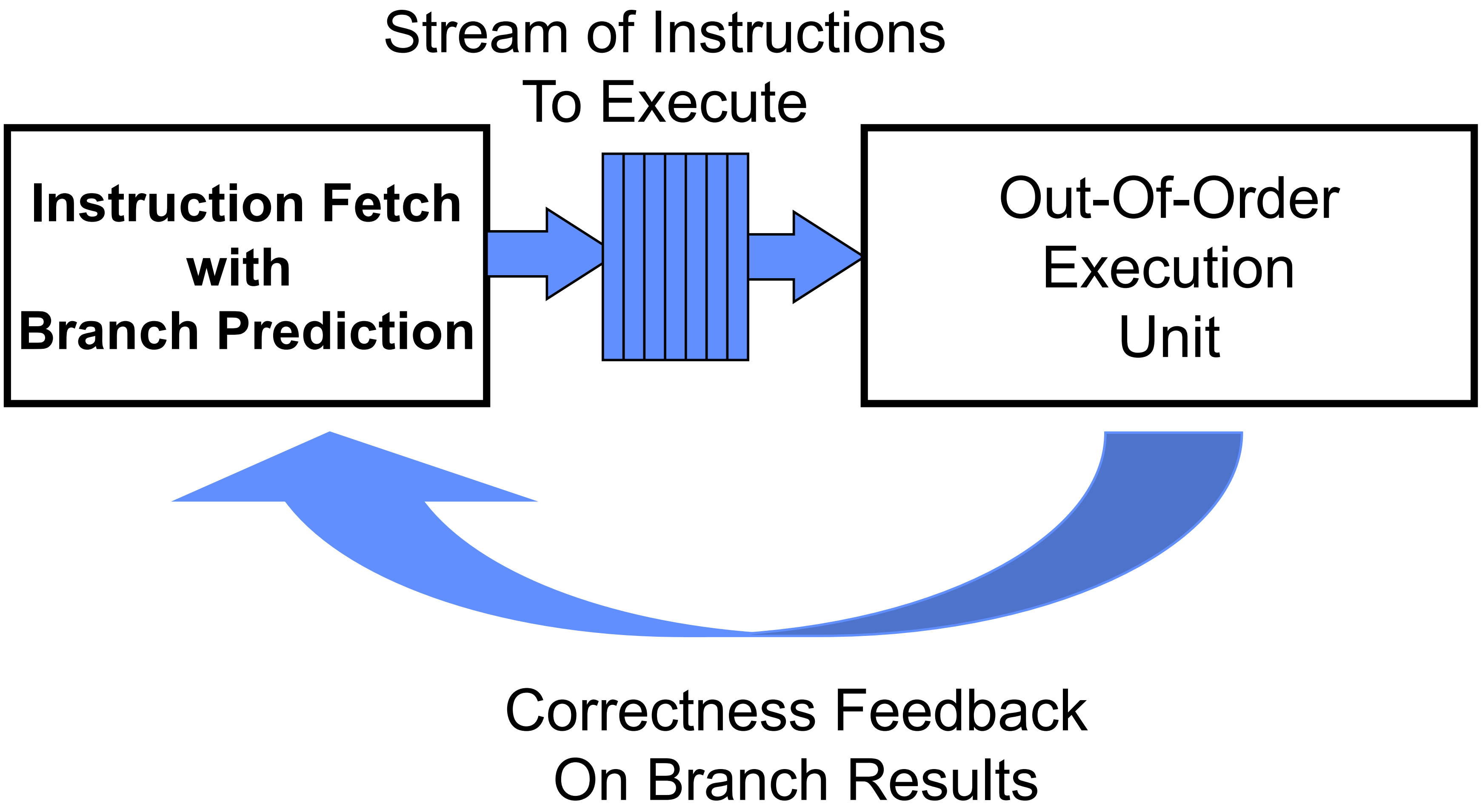


# Fetch Unit to ID|EXE Unit


Stream of Instructions
Instruction Fetch with Branch Prediction
To Execute
Out-Of-Order Execution Unit
Correctness Feedback On Branch Results

# Independent "Fetch" unit

- **Instruction fetch decoupled from execution**
  - **Instruction Buffer in-between**
- **Often issue logic (+ rename) included with Fetch**



## Explicit Register Renaming

## Register Renaming Summary

- Purpose of Renaming: removing “Anti-dependencies”
  - Get rid of WAR and WAW hazards, since these are not “real” dependencies
- Implicit Renaming: i.e. Tomasulo
  - Registers changed into values or response tags
  - We call this “implicit” because space in register file may or may not be used by results!
- Explicit Renaming: more physical registers than needed by ISA.
  - Rename table: tracks current association between architectural registers and physical registers
  - Uses a translation table to perform compiler-like transformation on the fly
- With Explicit Renaming:
  - All registers concentrated in single register file
  - Can utilize bypass network that looks more like 5-stage pipeline
  - Introduces a register-allocation problem
    - » Need to handle branch misprediction and precise exceptions differently, but ultimately makes things simpler

## Explicit Register Renaming

- Tomasulo provides *Implicit Register Renaming*
  - User registers renamed to reservation station tags
- Explicit Register Renaming:
  - Use *physical* register file that is larger than number of registers specified by ISA
- Keep a translation table:
  - ISA register => physical register mapping
  - When register is written, replace table entry with new register from freelist.
  - Physical register becomes free when not being used by any instructions in progress.
- Pipeline can be exactly like “standard” DLX pipeline
  - IF, ID, EX, etc….
- Advantages:
  - Removes all WAR and WAW hazards
  - Like Tomasulo, good for allowing full out-of-order completion
  - Allows data to be fetched from a single register file
  - Makes speculative execution/precise interrupts easier:
    - All that needs to be “undone” for precise break point is to undo the table mappings

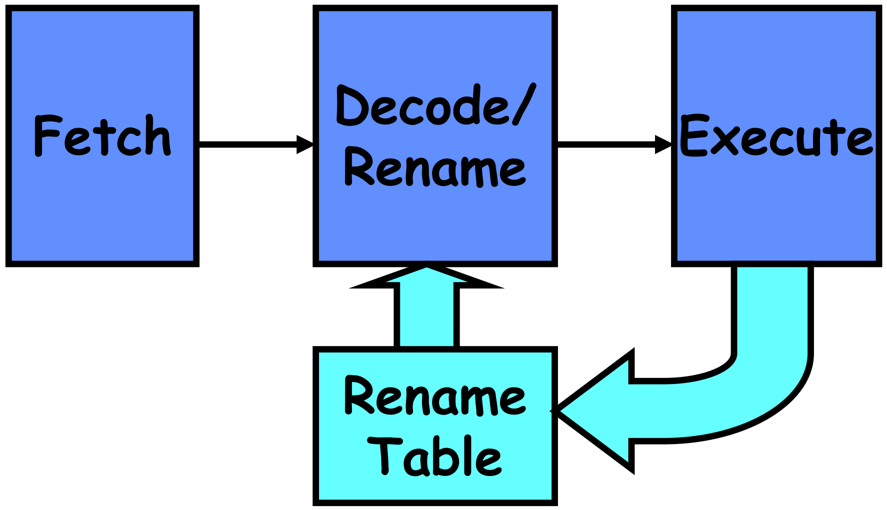
## Explicit Renaming Support Includes:

- Rapid access to a table of translations
- A physical register file that has more registers than specified by the ISA
- Ability to figure out which physical registers are free.
  - No free registers ⇒ stall on issue
- Thus, register renaming doesn't require reservation stations.

- Many modern architectures use explicit register renaming + Tomasulo-like reservation stations to control execution.
  - R10000, Alpha 21264, HP PA8000

## Explicit Register Renaming

- Make use of a *physical* register file that is larger than number of registers specified by ISA
- Keep a translation table:
  - ISA register => physical register mapping
  - When register is written, replace table entry with new register from freelist.
  - Physical register becomes free when not being used by any instructions in progress.

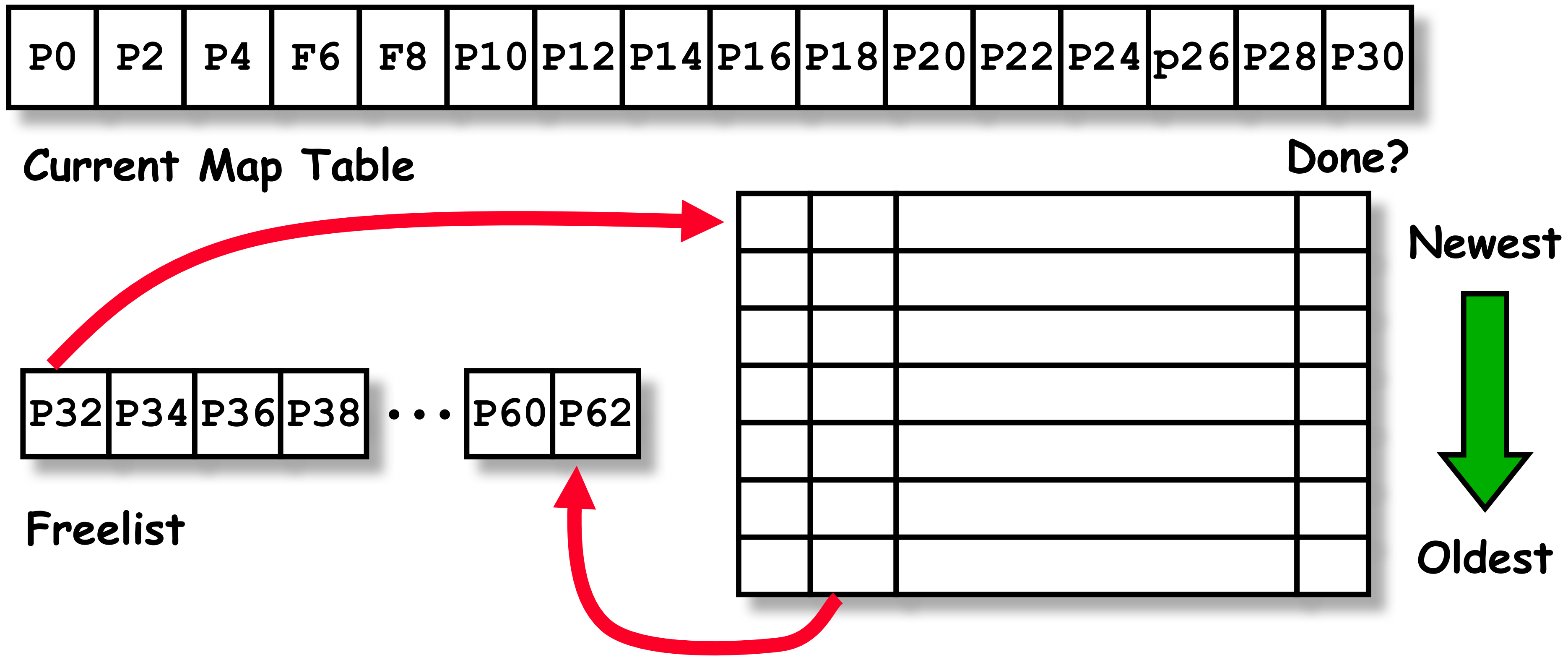


# Advantages of Explicit Renaming

- Decouples *renaming* from *scheduling:*
  - Pipeline can be exactly like "standard" DLX pipeline (perhaps with multiple operations issued per cycle)
  - Or, pipeline could be tomasulo-like or a scoreboard, etc.
  - Standard forwarding or bypassing could be used
- Allows data to be fetched from single register file
  - No need to bypass values from reorder buffer
  - This can be important for balancing pipeline
- Many processors use a variant of this technique:
  - R10000, Alpha 21264, HP PA8000
- Another way to get precise interrupt points:
  - All that needs to be "undone" for precise break point is to undo the table mappings
  - Provides an interesting mix between reorder buffer and future file
    - » Results are written immediately back to register file
    - » Registers *names* are "freed" in program order (by ROB)

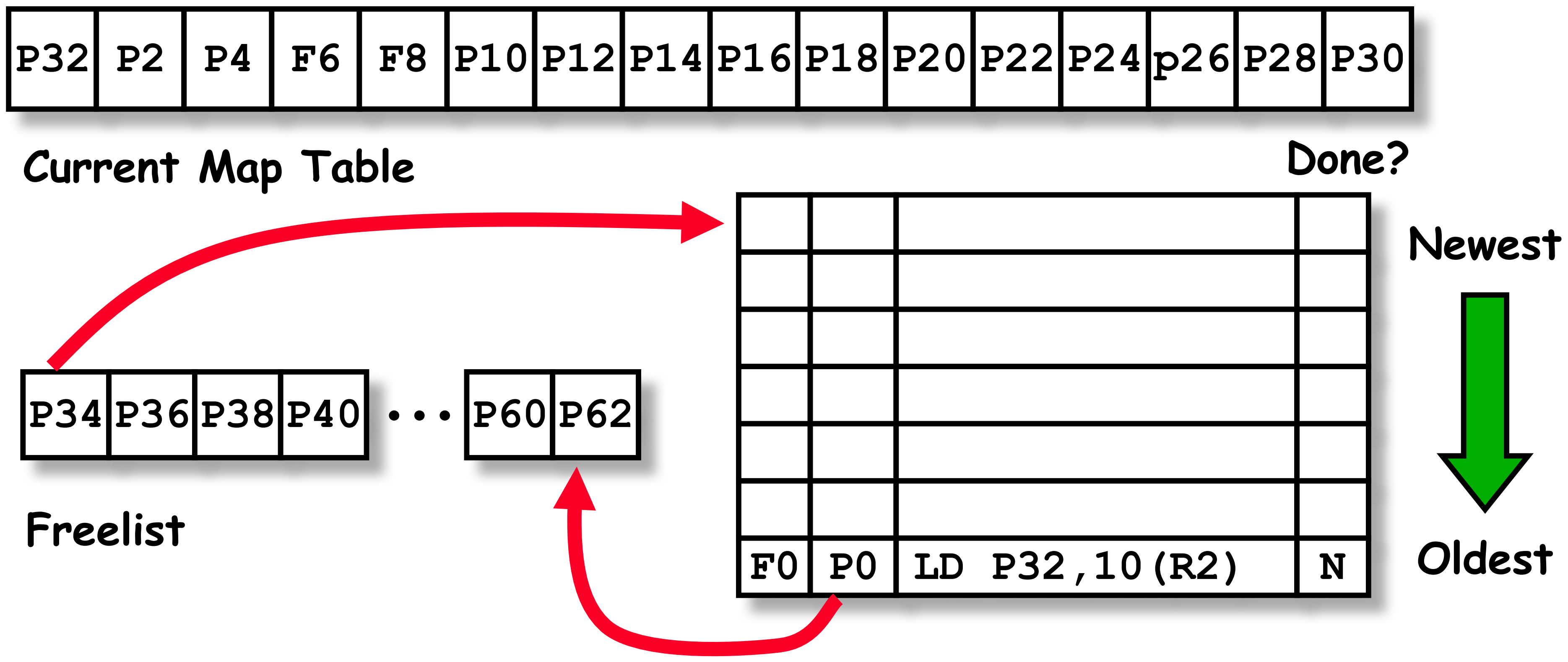
# Explicit register renaming:

## R10000 Freelist Management



- Physical register file larger than ISA register file
- On issue, each instruction that modifies a register is allocated new physical register from freelist
- Used on: R10000, Alpha 21264, HP PA8000

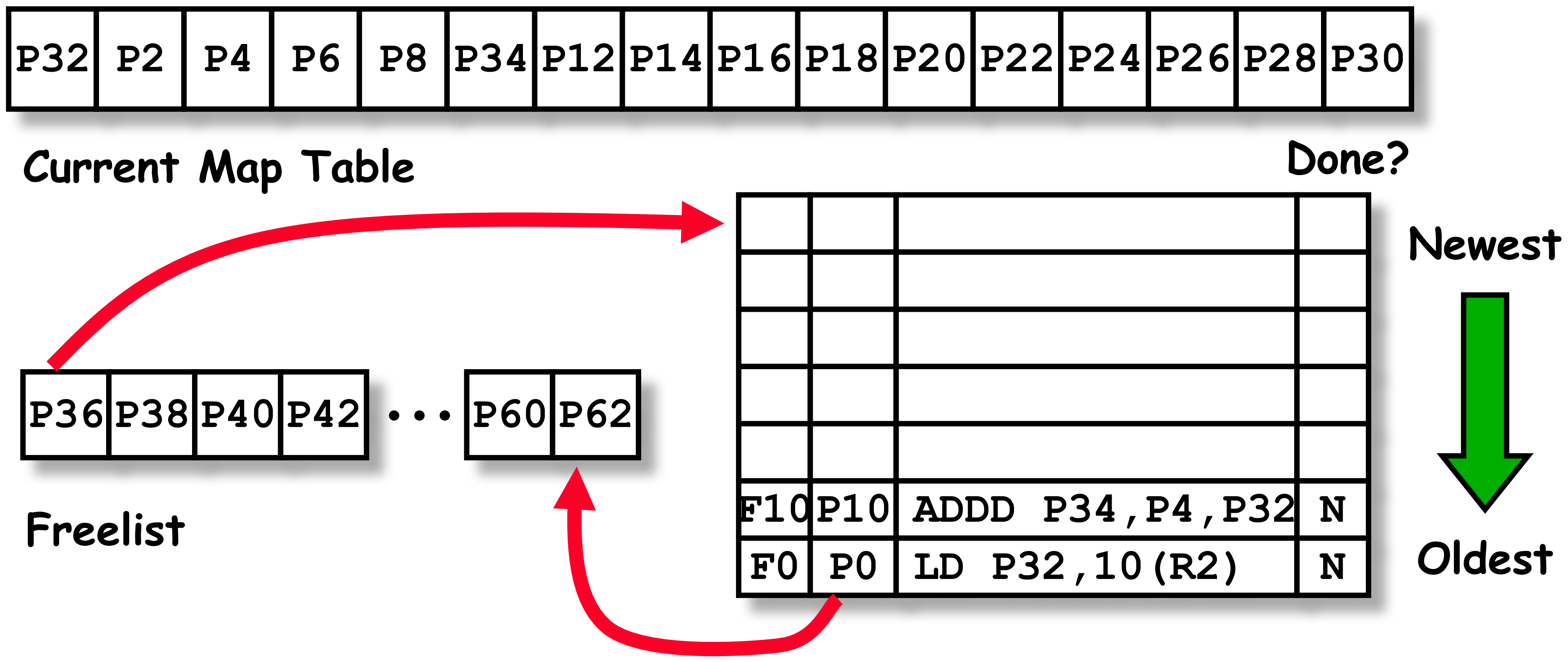
# Explicit register renaming:
## R10000 Freelist Management



P32 P2 P4 F6 F8 P10 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
Current Map Table
Done?
Newest
Oldest
P34 P36 P38 P40 ··· P60 P62
Freelist
F0 P0 LD P32,10(R2) N

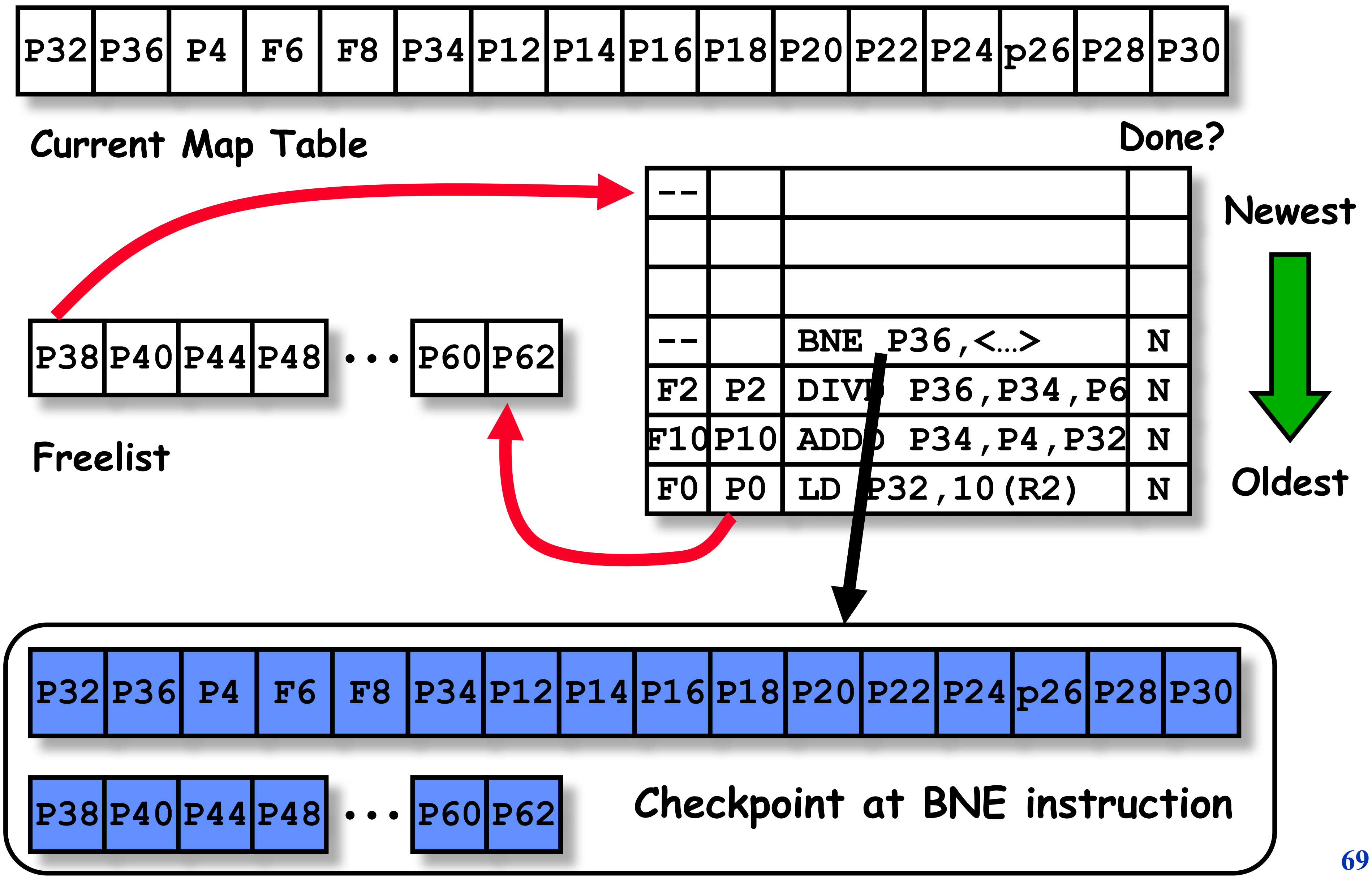
- Note that physical register P0 is "dead" (or not "live") past the point of this load.
  - When we go to commit the load, we free up

# Explicit register renaming:
## R10000 Freelist Management



P32 P2 P4 P6 P8 P34 P12 P14 P16 P18 P20 P22 P24 P26 P28 P30
Current Map Table
Done?
Newest
Oldest
P36 P38 P40 P42 ··· P60 P62
Freelist
F10 P10 ADDD P34,P4,P32 N
F0 P0 LD P32,10(R2) N

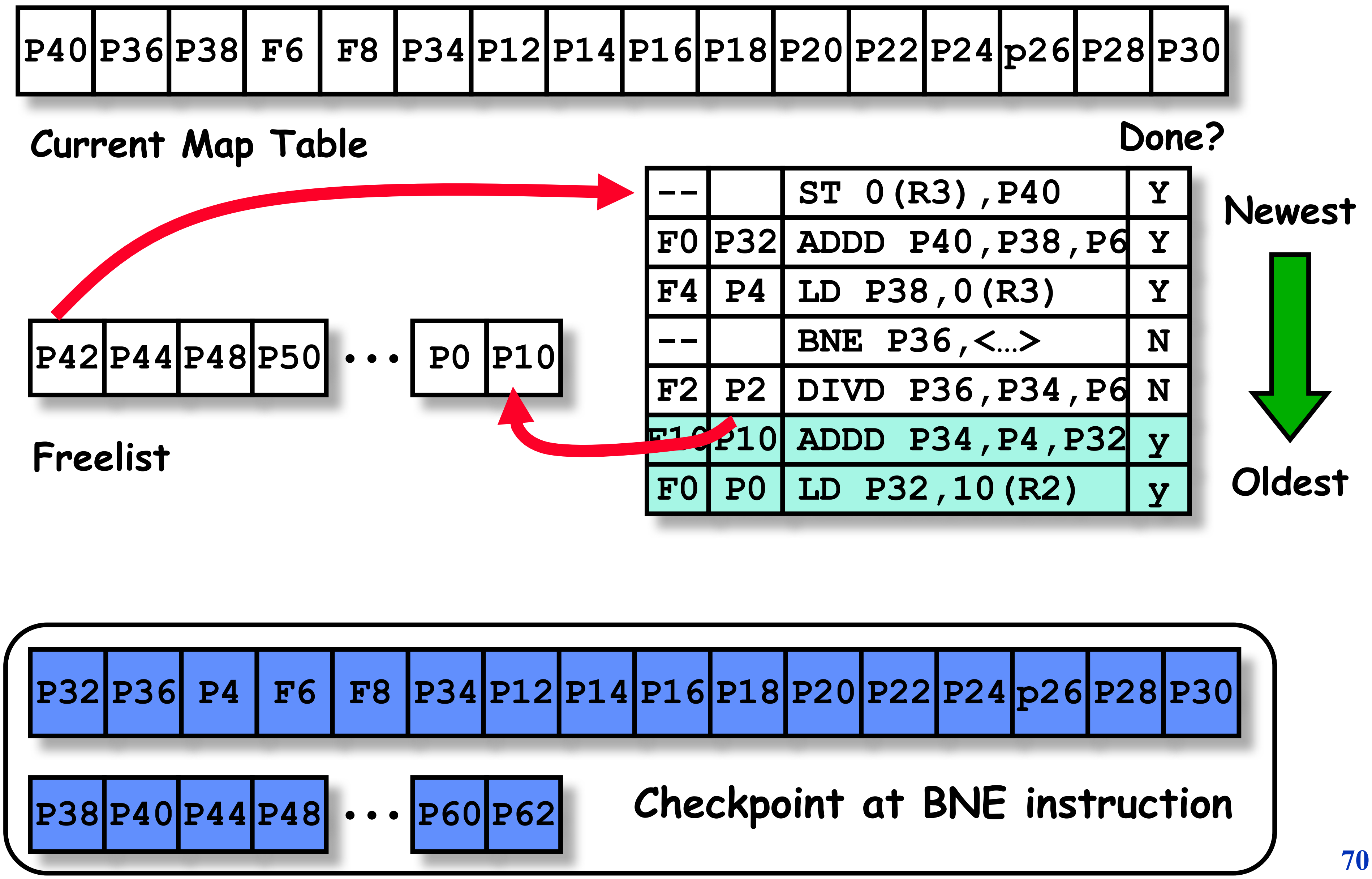
# Explicit register renaming:
## R10000 Freelist Management



P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
Current Map Table
Done?
Newest
Oldest
P38 P40 P44 P48 ••• P60 P62
Freelist
--
-- BNE P36,<…> N
F2 P2 DIVD P36,P34,P6 N
F10 P10 ADDD P34,P4,P32 N
F0 P0 LD P32,10(R2) N
P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
P38 P40 P44 P48 ••• P60 P62
Checkpoint at BNE instruction

# Explicit register renaming:

## R10000 Freelist Management



Current Map Table

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P40 | P36 | P38 | F6 | F8 | P34 | P12 | P14 | P16 | P18 | P20 | P22 | P24 | p26 | P28 | P30 |

Freelist

| | | | | | | |
|---|---|---|---|---|---|---|
| P42 | P44 | P48 | P50 | • • • | P0 | P10 |

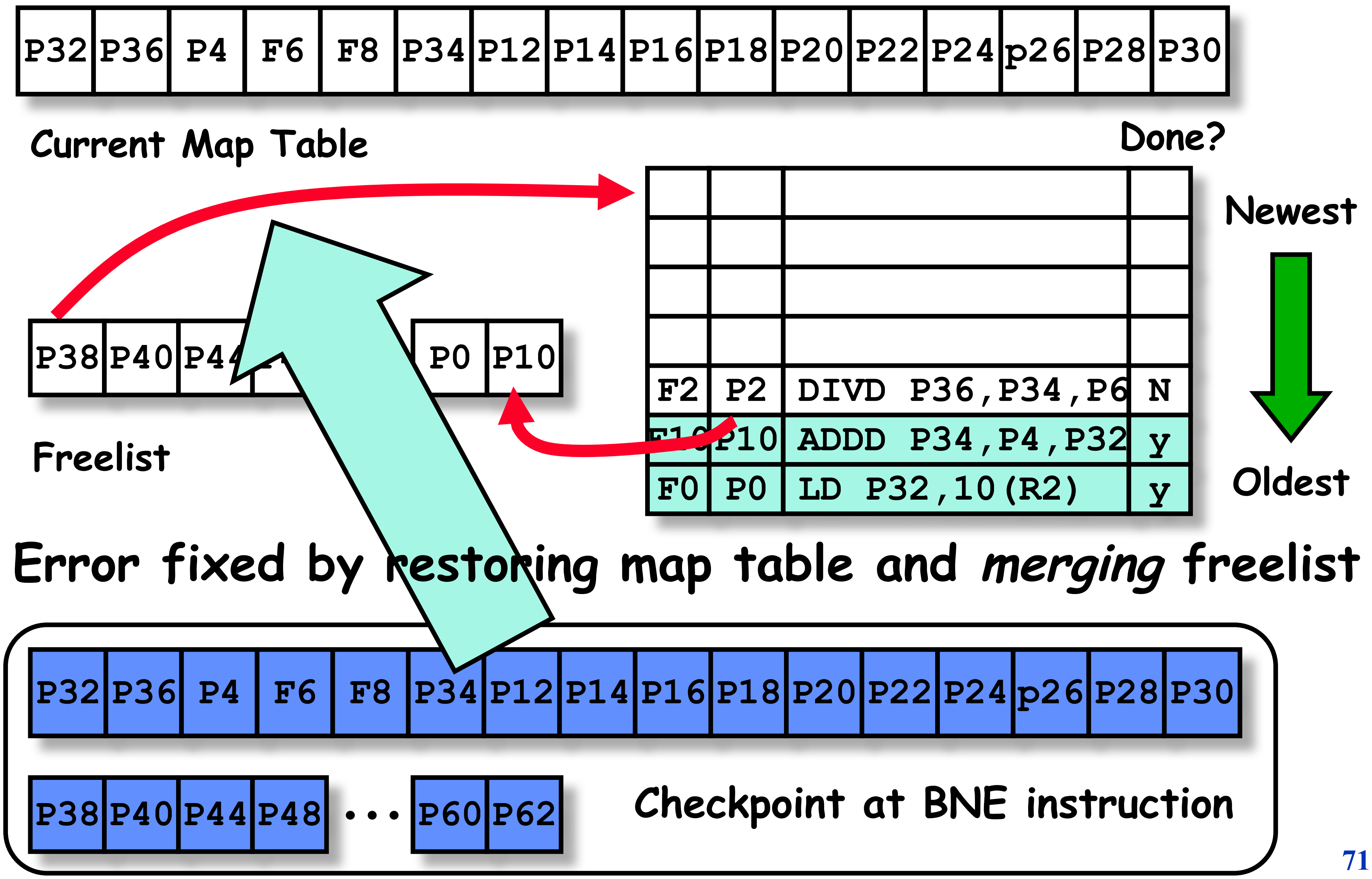
| | | | Done? |
|---|---|---|---|
| -- | | ST 0(R3),P40 | Y |
| F0 | P32 | ADDD P40,P38,P6 | Y |
| F4 | P4 | LD P38,0(R3) | Y |
| -- | | BNE P36,<…> | N |
| F2 | P2 | DIVD P36,P34,P6 | N |
| F10 | P10 | ADDD P34,P4,P32 | y |
| F0 | P0 | LD P32,10(R2) | y |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P32 | P36 | P4 | F6 | F8 | P34 | P12 | P14 | P16 | P18 | P20 | P22 | P24 | p26 | P28 | P30 |

| | | | | | | |
|---|---|---|---|---|---|---|
| P38 | P40 | P44 | P48 | • • • | P60 | P62 |

Checkpoint at BNE instruction

# Explicit register renaming:
## R10000 Freelist Management

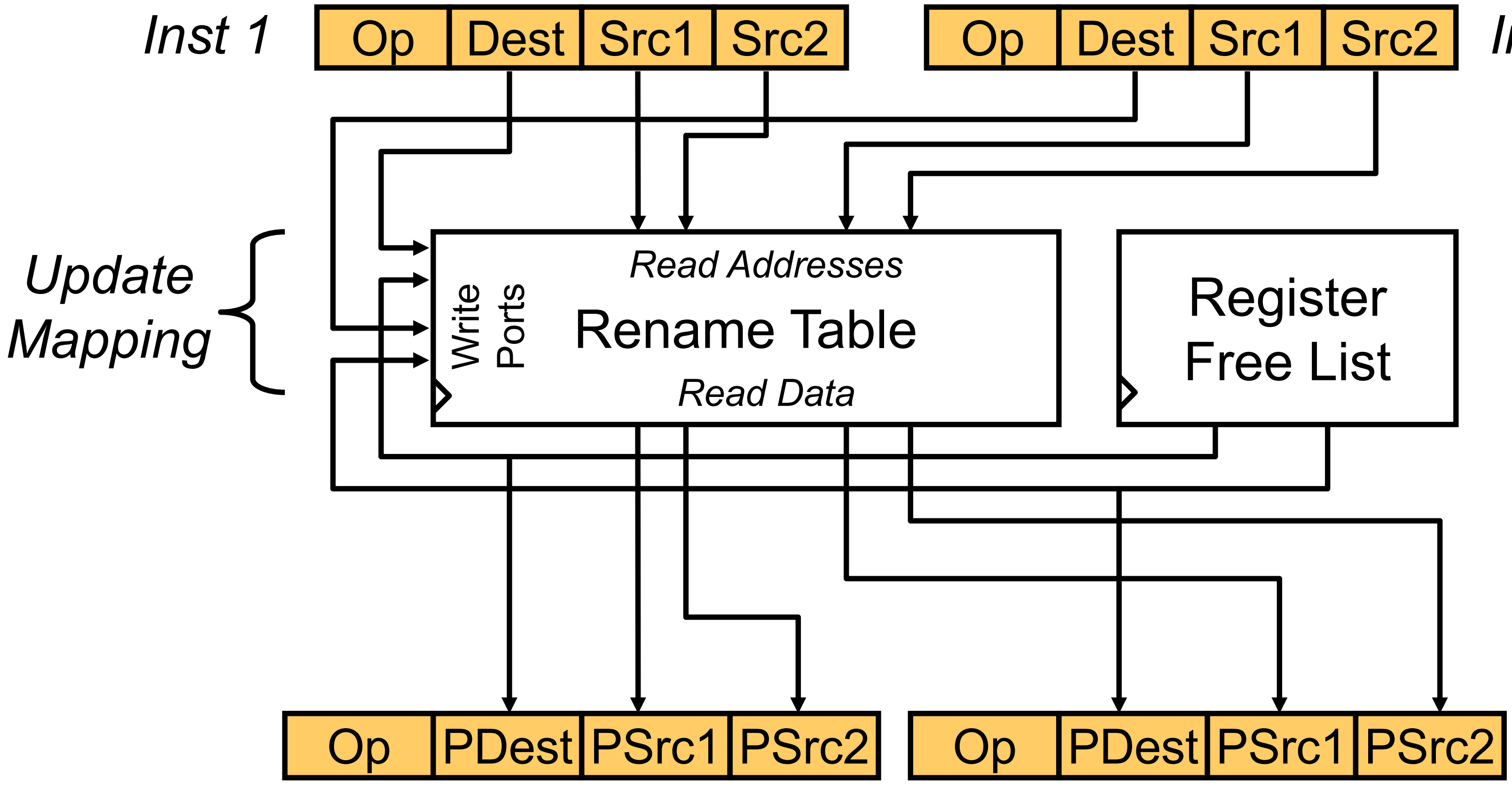


P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
Current Map Table
Done?
Newest
Oldest
P38 P40 P44 P0 P10
Freelist
F2 P2 DIVD P36,P34,P6 N
F10 P10 ADDD P34,P4,P32 y
F0 P0 LD P32,10(R2) y
P32 P36 P4 F6 F8 P34 P12 P14 P16 P18 P20 P22 P24 p26 P28 P30
P38 P40 P44 P48 ... P60 P62
Checkpoint at BNE instruction

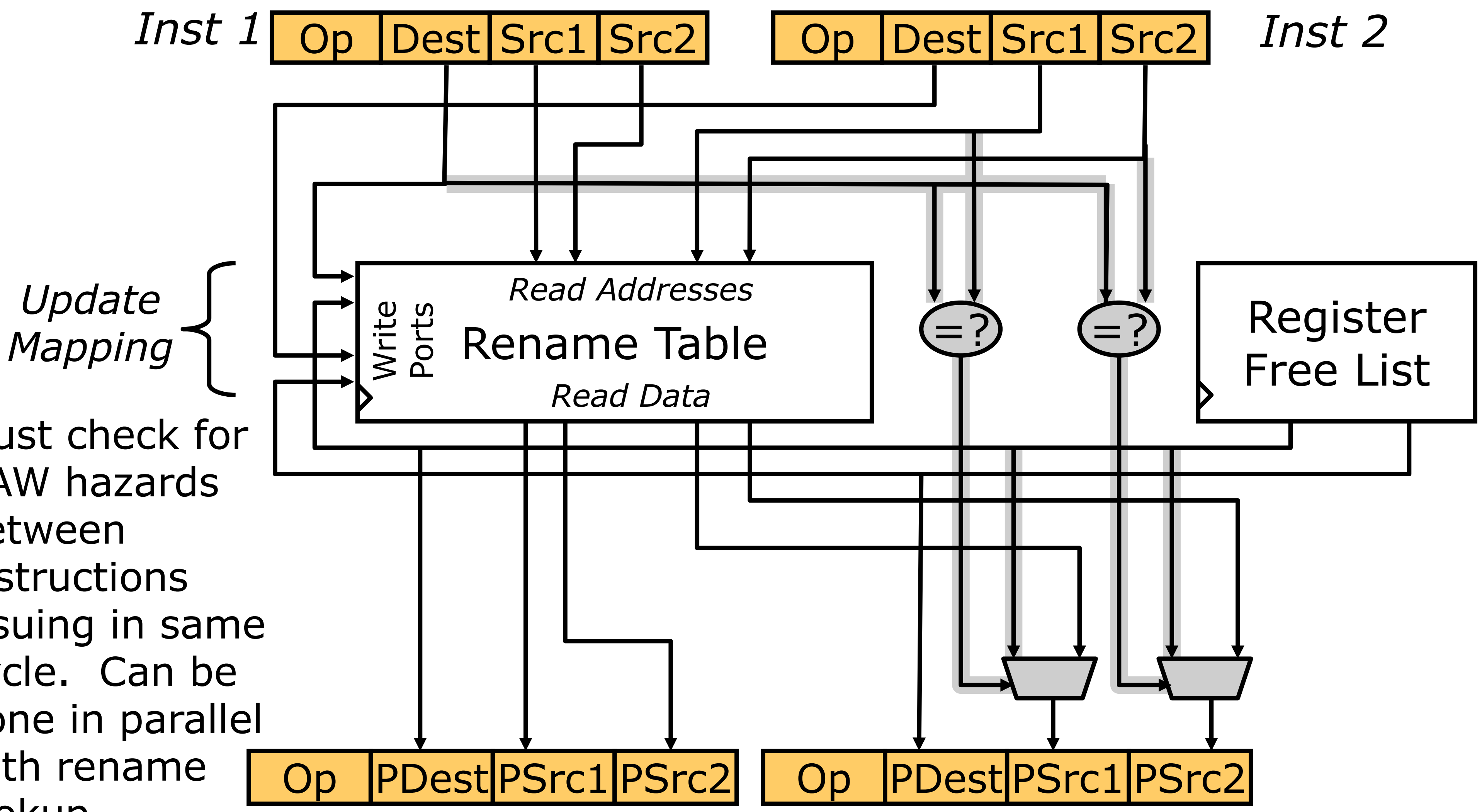
Error fixed by restoring map table and *merging* freelist

# Superscalar Register Renaming

- During decode, instructions allocated new physical destination register
- Source operands renamed to physical register with newest value
- Execution unit only sees physical register numbers



# Superscalar Register Renaming (Try #2)



Must check for RAW hazards between instructions issuing in same cycle. Can be done in parallel with rename lookup.

*MIPS R10K renames 4 serially-RAW-dependent insts/cycle*