# Lecture 13: Super-Scalar + Out-of-Order

## CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

## Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS252 at ETHZ
    - https://safari.ethz.ch/digitaltechnik/spring2023
  - CIS510 at Upenn
    - https://www.cis.upenn.edu/~cis5710/spring2024/

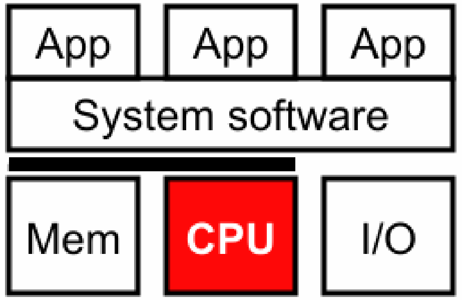
## Outline

- Super-scalar Processor
- Out-of-order Processor
  - Dynamic Scheduling in Hardware

## Parallelism

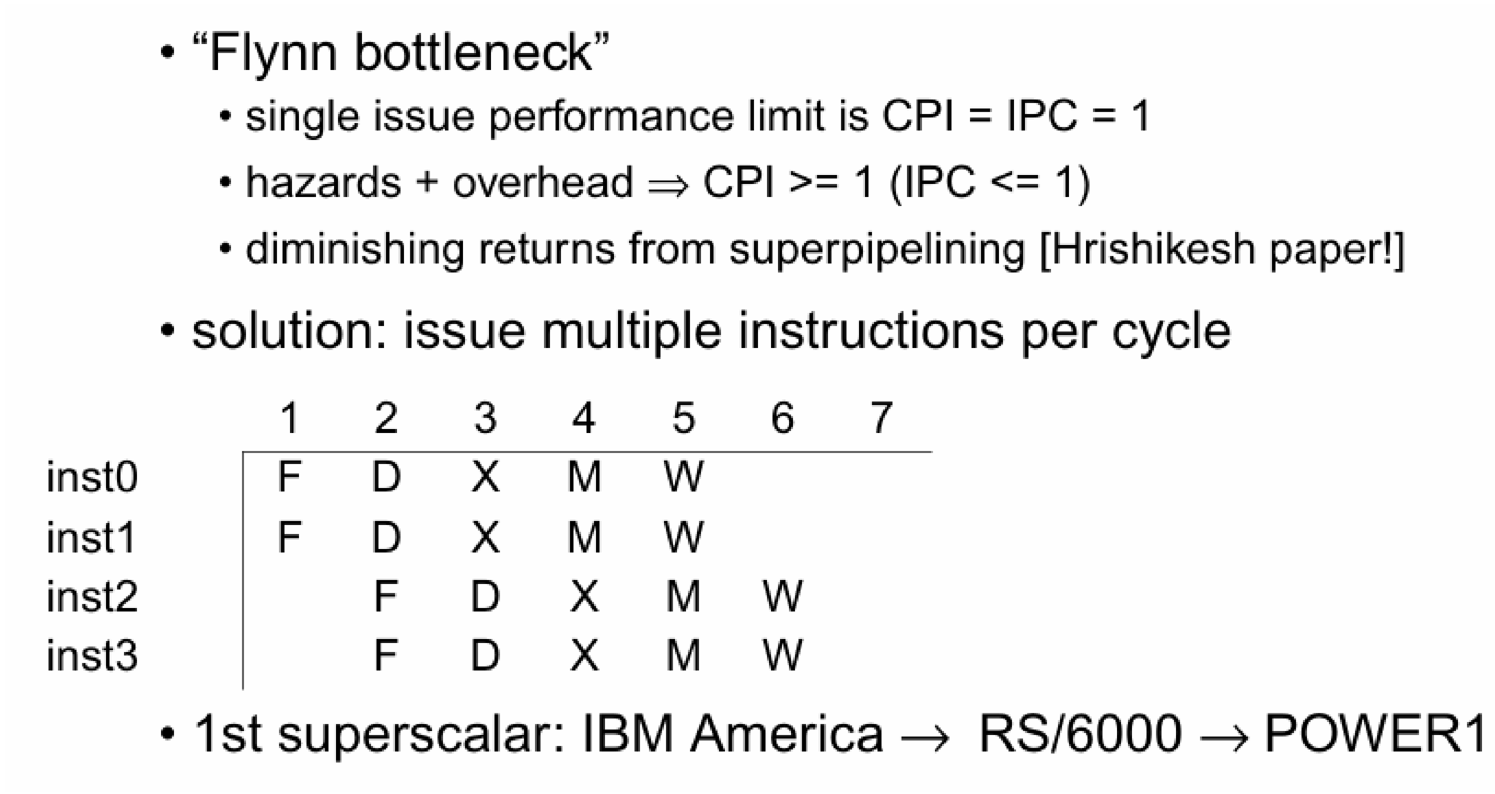
- Previously: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next
- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
- Then:
  - Static & dynamic scheduling
    - Extract much more ILP
  - Data-level parallelism (DLP)
    - Single-instruction, multiple data (one insn., four 64-bit adds)
  - Thread-level parallelism (TLP)
    - Multiple software threads running on multiple cores

## In-Order Super-scalar Pipelines



- Idea of instruction-level parallelism
- Superscalar hardware issues
  - Bypassing and register file
  - Stall logic
  - Fetch
- "Superscalar" vs VLIW/EPIC

## Flynn Bottleneck

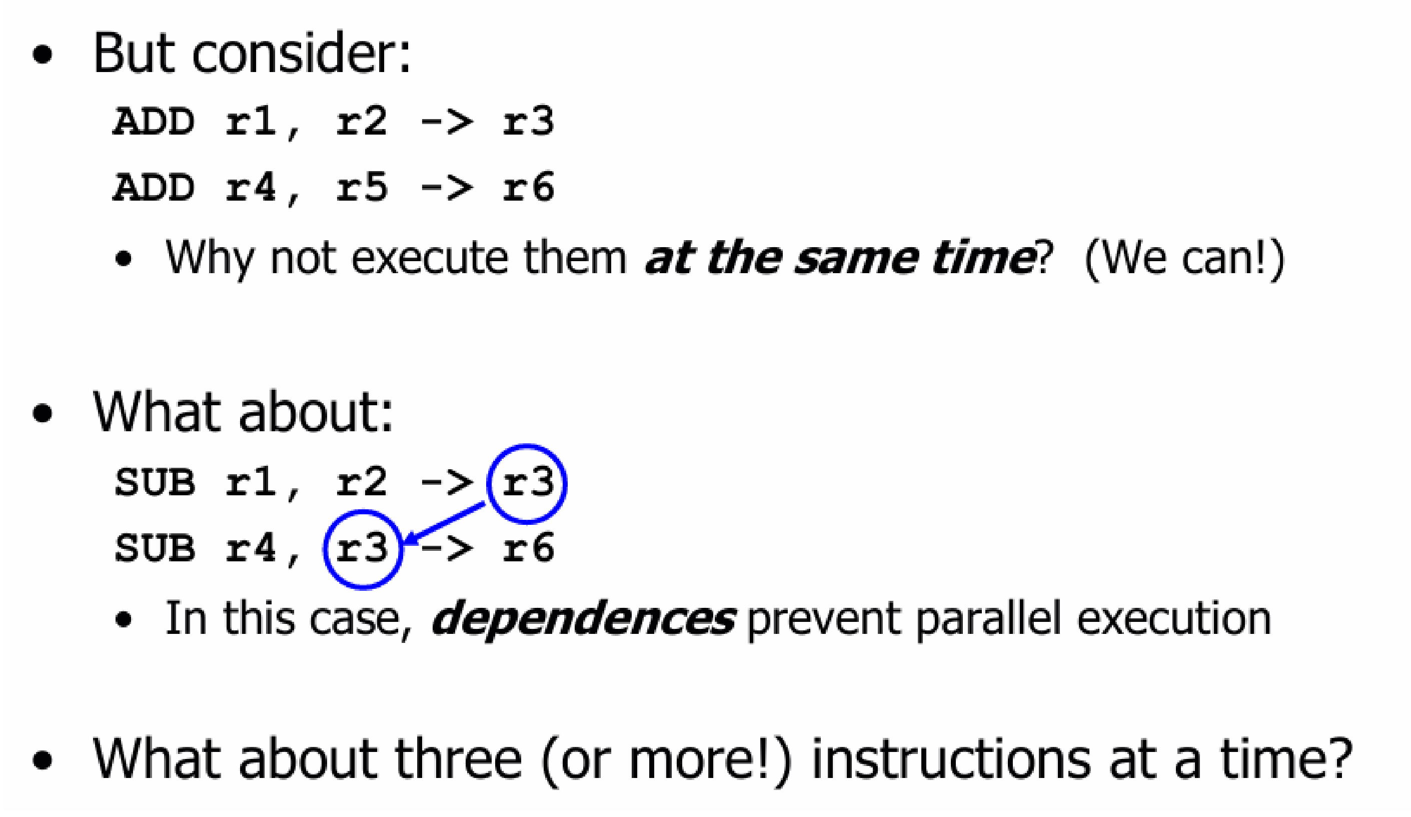


- “Flynn bottleneck”
  - single issue performance limit is CPI = IPC = 1
  - hazards + overhead ⇒ CPI >= 1 (IPC <= 1)
  - diminishing returns from superpipelining [Hrishikesh paper!]
- solution: issue multiple instructions per cycle

| | 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

## Instruction-level Parallelism (ILP)



- But consider:

```
ADD r1, r2 -> r3
ADD r4, r5 -> r6
```

  - Why not execute them ***at the same time***? (We can!)

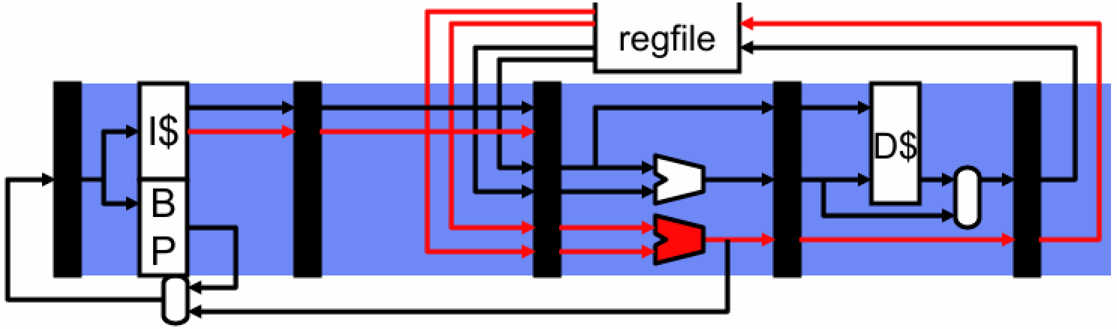
- What about:

```
SUB r1, r2 -> r3
SUB r4, r3 -> r6
```

  - In this case, ***dependences*** prevent parallel execution

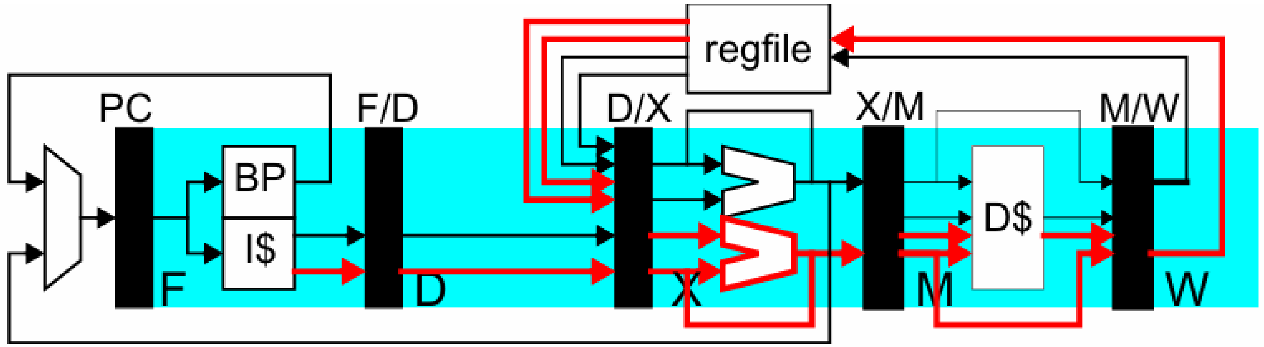
- What about three (or more!) instructions at a time?

# Multiple-Issue (Super-scalar) Pipeline



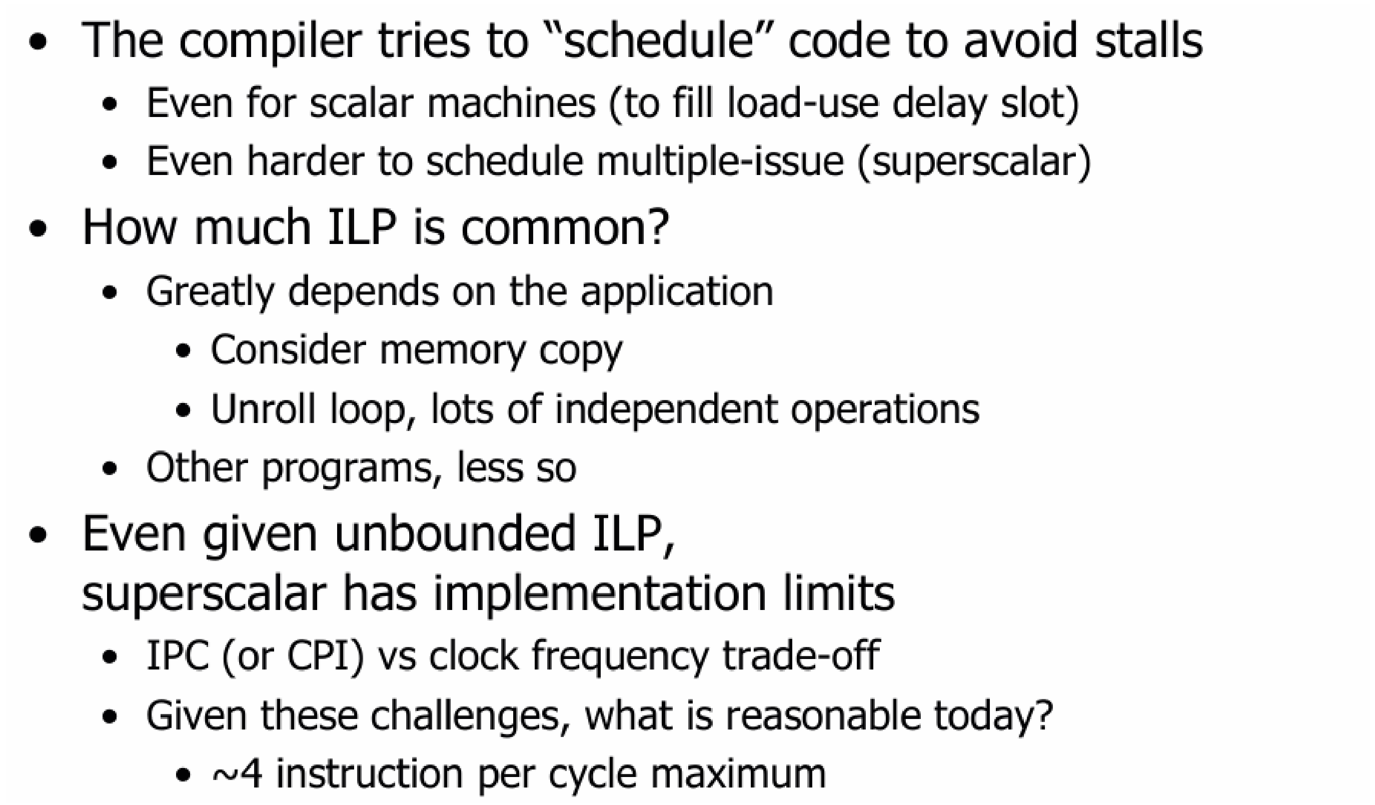
- Overcome this limit using **multiple issue**
  - Also called **superscalar**
  - Two instructions per stage at once, or three, or four, or eight…
  - **"Instruction-Level Parallelism (ILP)"** [Fisher, IEEE TC'81]
- Today, typically 4-6 wide (AMD, ARM, Intel)
  - AMD Zen 3 is 4-wide
  - Intel Golden Cove is 6-wide

# 5-Stage Dual-Issue 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?

## How Much ILP is There?



- The compiler tries to “schedule” code to avoid stalls
  - Even for scalar machines (to fill load-use delay slot)
  - Even harder to schedule multiple-issue (superscalar)
- How much ILP is common?
  - Greatly depends on the application
    - Consider memory copy
    - Unroll loop, lots of independent operations
  - Other programs, less so
- Even given unbounded ILP,
  superscalar has implementation limits
  - IPC (or CPI) vs clock frequency trade-off
  - Given these challenges, what is reasonable today?
    - ~4 instruction per cycle maximum

## Super-scalar Pipeline Diagrams - Ideal

| scalar | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `lw 0(r1)→r2` | F | D | X | M | W | | | | | | | |
| `lw 4(r1)→r3` | | F | D | X | M | W | | | | | | |
| `lw 8(r1)→r4` | | | F | D | X | M | W | | | | | |
| `add r14,r15→r6` | | | | F | D | X | M | W | | | | |
| `add r12,r13→r7` | | | | | F | D | X | M | W | | | |
| `add r17,r16→r8` | | | | | | F | D | X | M | W | | |
| `lw 0(r18)→r9` | | | | | | | F | D | X | M | W | |

| 2-way superscalar | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `lw 0(r1)→r2` | F | D | X | M | W | | | | | | | |
| `lw 4(r1)→r3` | F | D | X | M | W | | | | | | | |
| `lw 8(r1)→r4` | | F | D | X | M | W | | | | | | |
| `add r14,r15→r6` | | F | D | X | M | W | | | | | | |
| `add r12,r13→r7` | | | F | D | X | M | W | | | | | |
| `add r17,r16→r8` | | | F | D | X | M | W | | | | | |
| `lw 0(r18)→r9` | | | | F | D | X | M | W | | | | |

# Super-scalar Stalls

- invariant: stalls propagate upstream to younger instructions
- what if older instruction in issue "pair" (inst0) stalls?
  - younger instruction (inst1) stalls too, cannot pass it
- what if younger instruction (inst1) stalls?
  - can older instruction from next group (inst2) move up?

rigid pipe ⇒ no

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| inst0 | F | D | X | M |
| inst1 | F | D | **d*** | X |
| inst2 | | F | **p*** | D |
| inst3 | | F | **p*** | D |

fluid pipe ⇒ yes

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| inst0 | F | D | X | M |
| inst1 | F | D | **d*** | X |
| inst2 | | F | **D** | X |
| inst3 | | F | **p*** | D |

# Super-scalar Pipeline Diagrams - Realistic

| scalar | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `lw 0(r1)→r2` | F | D | X | M | W | | | | | | | |
| `lw 4(r1)→r3` | | F | D | X | M | W | | | | | | |
| `lw 8(r1)→r4` | | | F | D | X | M | W | | | | | |
| `add r4,r5→r6` | | | | F | D | * | X | M | W | | | |
| `add r2,r3→r7` | | | | | F | * | D | X | M | W | | |
| `add r7,r6→r8` | | | | | | | F | D | X | M | W | |
| `lw 4(r8)→r9` | | | | | | | | F | D | X | M | W |

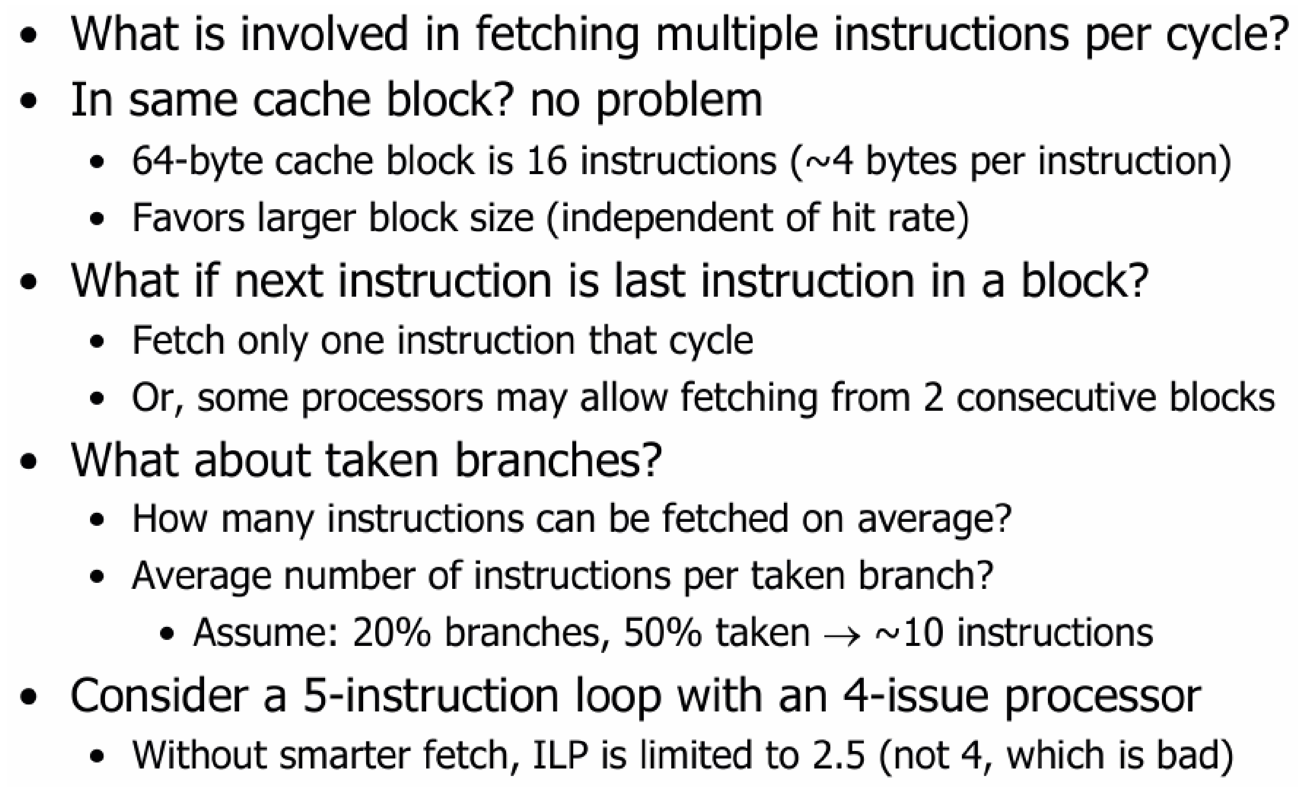
| 2-way superscalar | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `lw 0(r1)→r2` | F | D | X | M | W | | | | | | | |
| `lw 4(r1)→r3` | F | D | X | M | W | | | | | | | |
| `lw 8(r1)→r4` | | F | D | X | M | W | | | | | | |
| `add r4,r5→r6` | | F | D | * | * | X | M | W | | | | |
| `add r2,r3→r7` | | | F | * | * | D | X | M | W | | | |
| `add r7,r6→r8` | | | F | * | * | * | D | X | M | W | | |
| `lw 4(r8)→r9` | | | | F | * | * | * | D | X | M | W | |

Rigid pipe

## Super-scalar Challenges – Front End

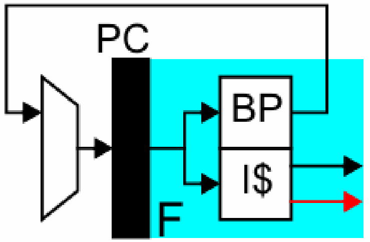
- **Superscalar instruction fetch**
  - Modest: fetch multiple instructions per cycle
  - Aggressive: buffer instructions and/or predict multiple branches
- **Superscalar instruction decode**
  - Replicate decoders
- **Superscalar instruction issue**
  - Determine when instructions can proceed in parallel
  - More complex stall logic - order $\mathbf{N^2}$ for $N$-wide machine
  - Not all combinations of types of instructions possible
- **Superscalar register read**
  - Port for each register read (4-wide superscalar ➔ 8 read "ports")
  - Each port needs its own set of address and data wires
    - Latency & area $\propto$ #ports$^2$

## Challenges of Super-scalar Fetch



- What is involved in fetching multiple instructions per cycle?
- In same cache block? no problem
  - 64-byte cache block is 16 instructions (~4 bytes per instruction)
  - Favors larger block size (independent of hit rate)
- What if next instruction is last instruction in a block?
  - Fetch only one instruction that cycle
  - Or, some processors may allow fetching from 2 consecutive blocks
- What about taken branches?
  - How many instructions can be fetched on average?
  - Average number of instructions per taken branch?
    - Assume: 20% branches, 50% taken → ~10 instructions
- Consider a 5-instruction loop with an 4-issue processor
  - Without smarter fetch, ILP is limited to 2.5 (not 4, which is bad)

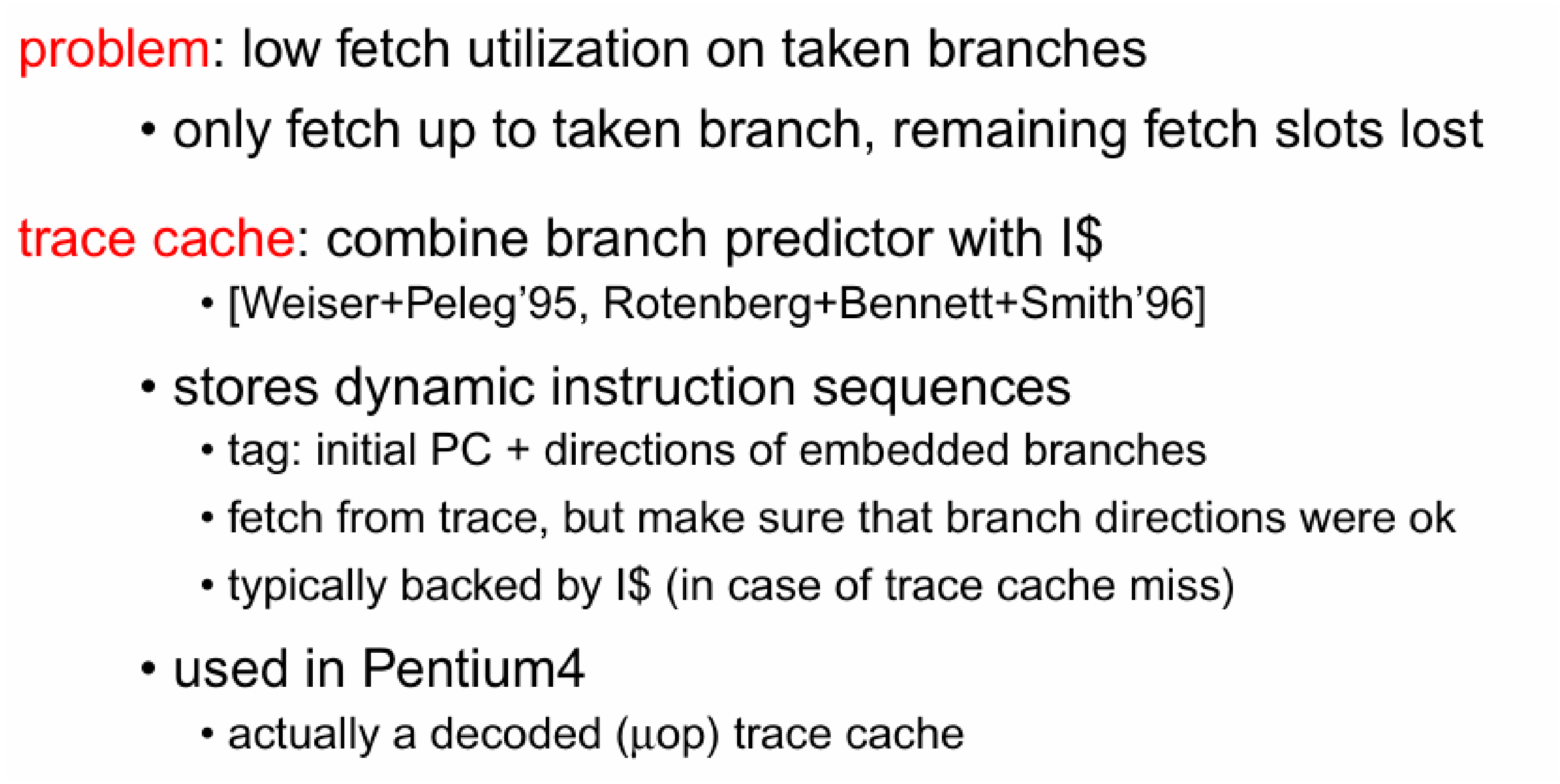
## Wide Fetch



what is involved in fetching multiple instructions per cycle?

- if instructions are sequential...
  - and on same cache line ⇒ nothing really
  - and on different cache lines ⇒ banked I$ + combining network
- if instructions are not sequential...
  - – more difficult
  - – two serial I$ accesses (access1⇒predict target⇒access2)? no
- note: embedded branches OK as long as predicted NT
  - serial access + prediction in parallel
  - if prediction is T, discard serial part after branch

## Trace Cache



problem: low fetch utilization on taken branches

- only fetch up to taken branch, remaining fetch slots lost

trace cache: combine branch predictor with I$

- [Weiser+Peleg'95, Rotenberg+Bennett+Smith'96]
- stores dynamic instruction sequences
  - tag: initial PC + directions of embedded branches
  - fetch from trace, but make sure that branch directions were ok
  - typically backed by I$ (in case of trace cache miss)
- used in Pentium4
  - actually a decoded (μop) trace cache

# Trace Cache Example

- instruction cache with 2 instrs per cache block

| I$ | |
|---|---|
| tag | data |
| PC0 | inst0,inst1 |
| PC2 | inst2,inst3 |
| PC4 | inst4,inst5 |

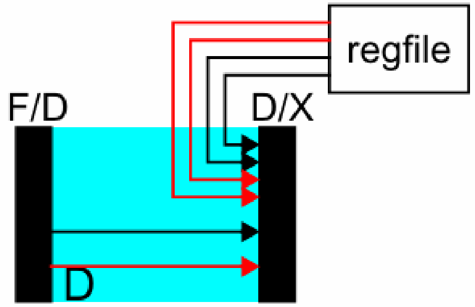
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| inst0 (beq r1, inst4) | F | D | X | M | W | | |
| inst4 | f* | F | D | X | M | W | |
| inst5 | | F | D | X | M | W | |
| inst6 | | | F | D | X | M | W |

- trace-cache with 2 instrs per cache block

| T$ | |
|---|---|
| tag | data |
| PC0:T | inst0,inst4 |
| PC2:– | inst2,inst3 |
| PC5:– | inst5,inst6 |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| inst0 (beq r1, inst4) | F | D | X | M | W | | |
| inst4 | F | D | X | M | W | | |
| inst5 | | F | D | X | M | W | |
| inst6 | | F | D | X | M | W | |

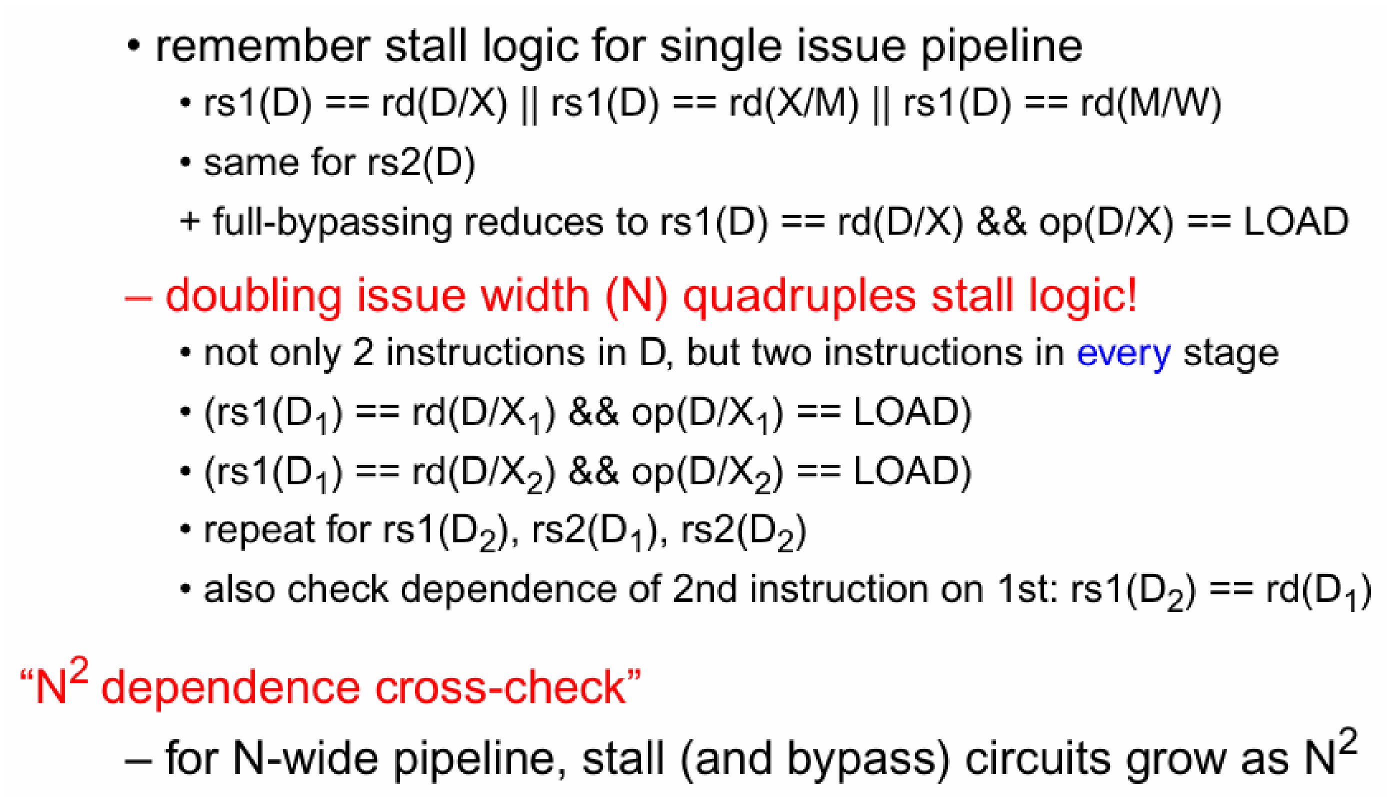
## Wide Decode



what is involved in decoding N instructions per cycle?

- actually decoding instructions?
  - + easy if fixed length instructions (multiple decoders)
  - – harder (but possible) if variable length
- reading input register values?
  - – 2N register read ports (register file read latency ~2N)
  - • actually less than 2N, since most values come from bypasses
- what about the stall logic to enforce RAW dependences?

## $N^2$ Dependence Cross-Check



- remember stall logic for single issue pipeline
  - rs1(D) == rd(D/X) || rs1(D) == rd(X/M) || rs1(D) == rd(M/W)
  - same for rs2(D)

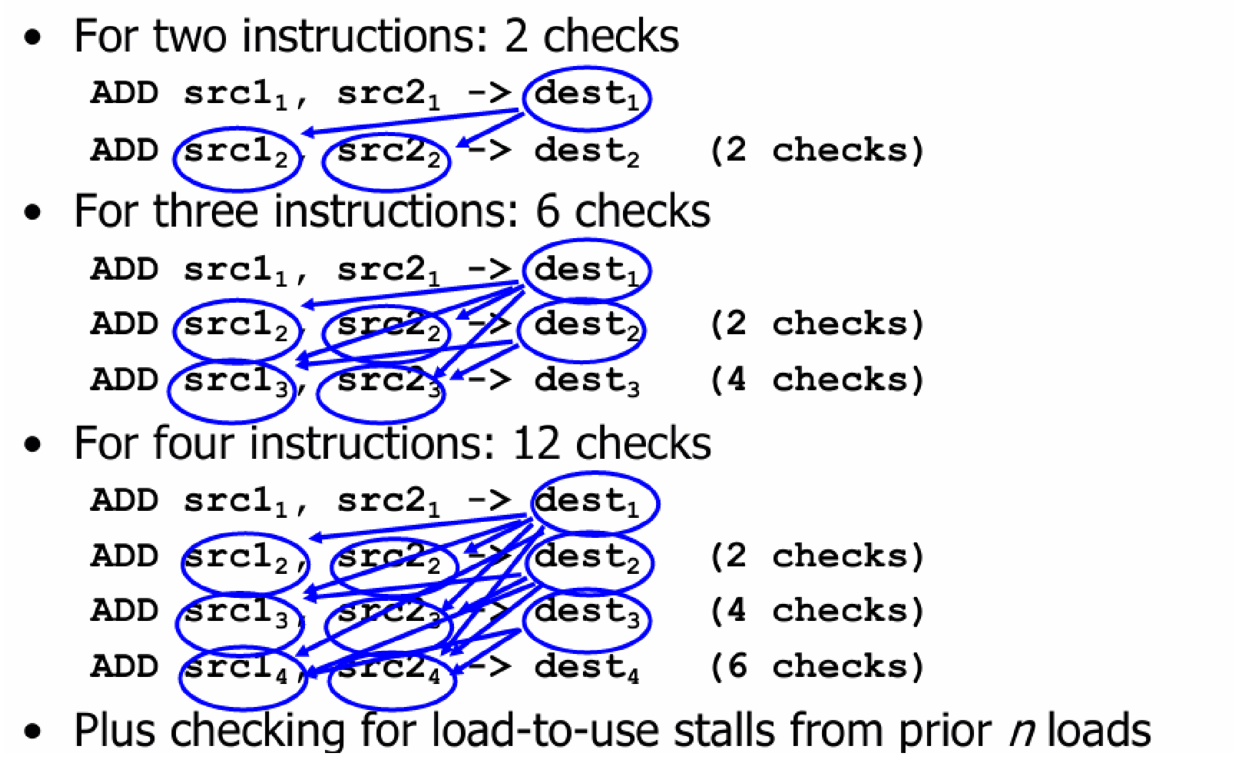
  + full-bypassing reduces to rs1(D) == rd(D/X) && op(D/X) == LOAD

– doubling issue width (N) quadruples stall logic!
  - not only 2 instructions in D, but two instructions in every stage
  - (rs1($D_1$) == rd($D/X_1$) && op($D/X_1$) == LOAD)
  - (rs1($D_1$) == rd($D/X_2$) && op($D/X_2$) == LOAD)
  - repeat for rs1($D_2$), rs2($D_1$), rs2($D_2$)
  - also check dependence of 2nd instruction on 1st: rs1($D_2$) == rd($D_1$)

"$N^2$ dependence cross-check"

– for N-wide pipeline, stall (and bypass) circuits grow as $N^2$

## What Checking Is Required?



- For two instructions: 2 checks

ADD $\text{src1}_1$, $\text{src2}_1$ -> $\text{dest}_1$
ADD $\text{src1}_2$, $\text{src2}_2$ -> $\text{dest}_2$ (2 checks)

- For three instructions: 6 checks

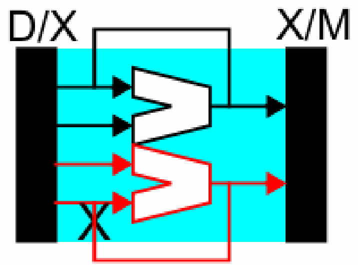
ADD $\text{src1}_1$, $\text{src2}_1$ -> $\text{dest}_1$
ADD $\text{src1}_2$, $\text{src2}_2$ -> $\text{dest}_2$ (2 checks)
ADD $\text{src1}_3$, $\text{src2}_3$ -> $\text{dest}_3$ (4 checks)

- For four instructions: 12 checks

ADD $\text{src1}_1$, $\text{src2}_1$ -> $\text{dest}_1$
ADD $\text{src1}_2$, $\text{src2}_2$ -> $\text{dest}_2$ (2 checks)
ADD $\text{src1}_3$, $\text{src2}_3$ -> $\text{dest}_3$ (4 checks)
ADD $\text{src1}_4$, $\text{src2}_4$ -> $\text{dest}_4$ (6 checks)

- Plus checking for load-to-use stalls from prior *n* loads

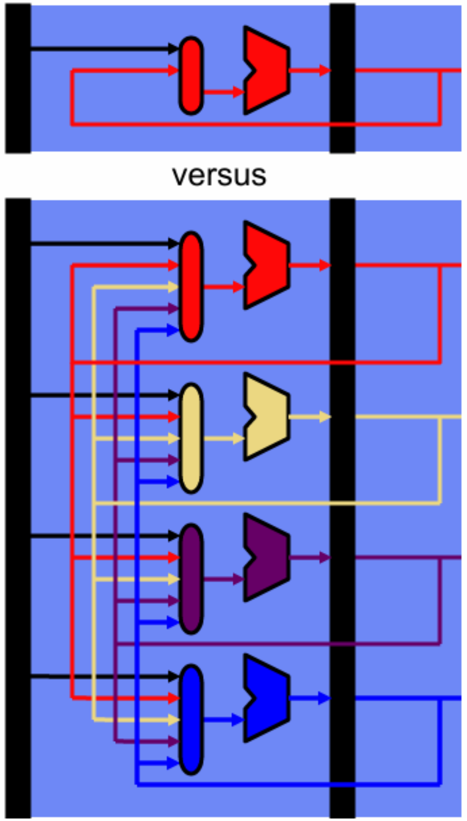
## Wide Execute



what is involved in execut]ing N instructions per cycle?

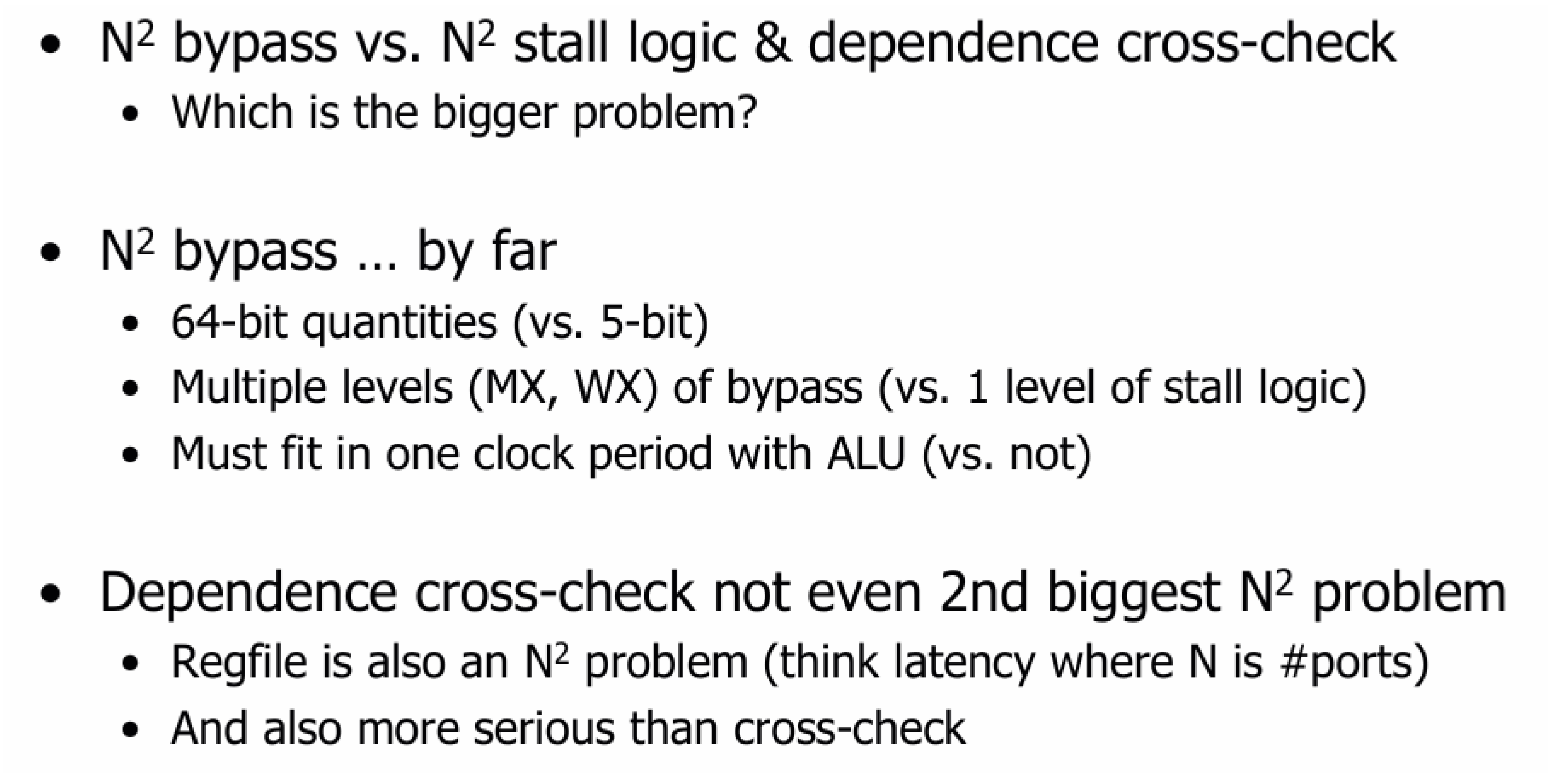
- multiple execution units…N of every kind?
  - N ALUs? OK, ALUs are small
  - N FP dividers? no, FP dividers are huge (and fdiv is uncommon)
- typically have some mix (proportional to instruction mix)
  - RS/6000: 1 ALU/memory/branch + 1 FP
  - Pentium: 1 any + 1 ALU (Pentium)
  - Pentium II: 1 ALU/FP + 1 ALU + 1 load + 1 store + 1 branch
  - Alpha 21164: 1 ALU/FP/branch + 2 ALU + 1 load/store

## $N^2$ Bypass



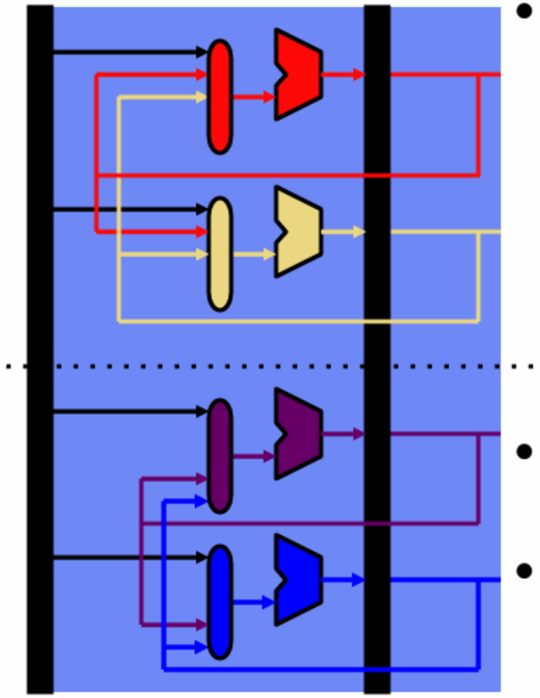
- **$N^2$ bypass network**
  - N+1 input muxes at each ALU input
  - $N^2$ point-to-point connections
  - Routing lengthens wires
  - Heavy capacitive load
- And this is just one bypass stage (MX)!
  - There is also WX bypassing
  - Even more for deeper pipelines
- One of the big problems of superscalar
  - Why? On the critical path of single-cycle "bypass & execute" loop

## Not All $N^2$ Created Equal



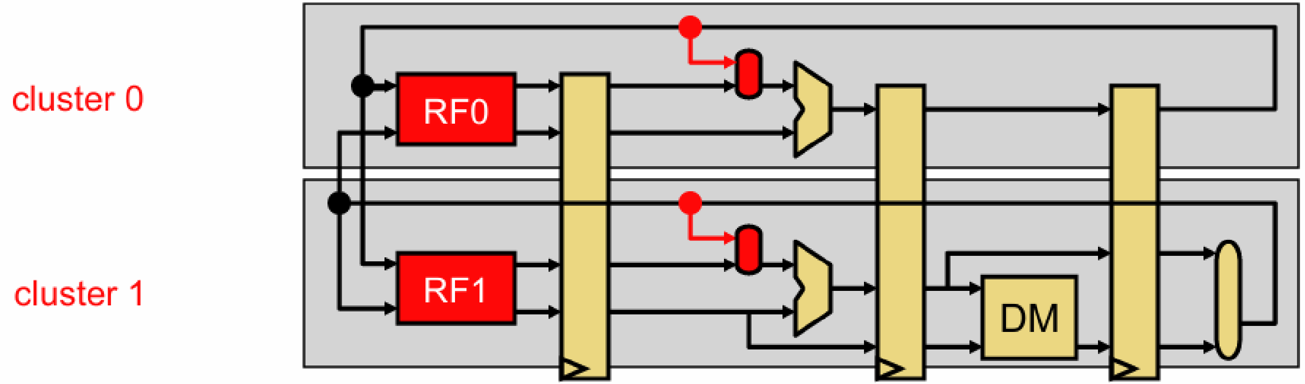
- $N^2$ bypass vs. $N^2$ stall logic & dependence cross-check
  - Which is the bigger problem?
- $N^2$ bypass … by far
  - 64-bit quantities (vs. 5-bit)
  - Multiple levels (MX, WX) of bypass (vs. 1 level of stall logic)
  - Must fit in one clock period with ALU (vs. not)
- Dependence cross-check not even 2nd biggest $N^2$ problem
  - Regfile is also an $N^2$ problem (think latency where N is #ports)
  - And also more serious than cross-check

## Mitigating N$^2$ Bypass & Register File



- **Clustering**: mitigates N$^2$ bypass
  - Group ALUs into **K** clusters
  - Full bypassing within a cluster
  - Limited bypassing between clusters
    - **With 1 or 2 cycle delay**
    - Can hurt IPC, but faster clock
  - (N/K) + 1 inputs at each mux
  - (N/K)$^2$ bypass paths in each cluster
- **Steering**: key to performance
  - Steer dependent insns to same cluster
- **Cluster register file**, too
  - Replicate a register file per cluster
  - All register writes update all replicas
  - Fewer read ports; only for cluster

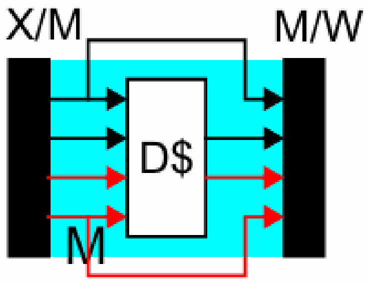
## Mitigating $N^2$ RegFile with Clustering



- **Clustering**: split **N**-wide execution pipeline into **K** clusters
  - With centralized register file, 2N read ports and N write ports

- **Clustered register file**: extend clustering to register file
  - Replicate the register file (one replica per cluster)
  - Register file supplies register operands to just its cluster
  - All register writes go to all register files (keep them in sync)
  - Advantage: fewer read ports per register!
    - K register files, each with 2N/K read ports and N write ports

## Wide Memory Access



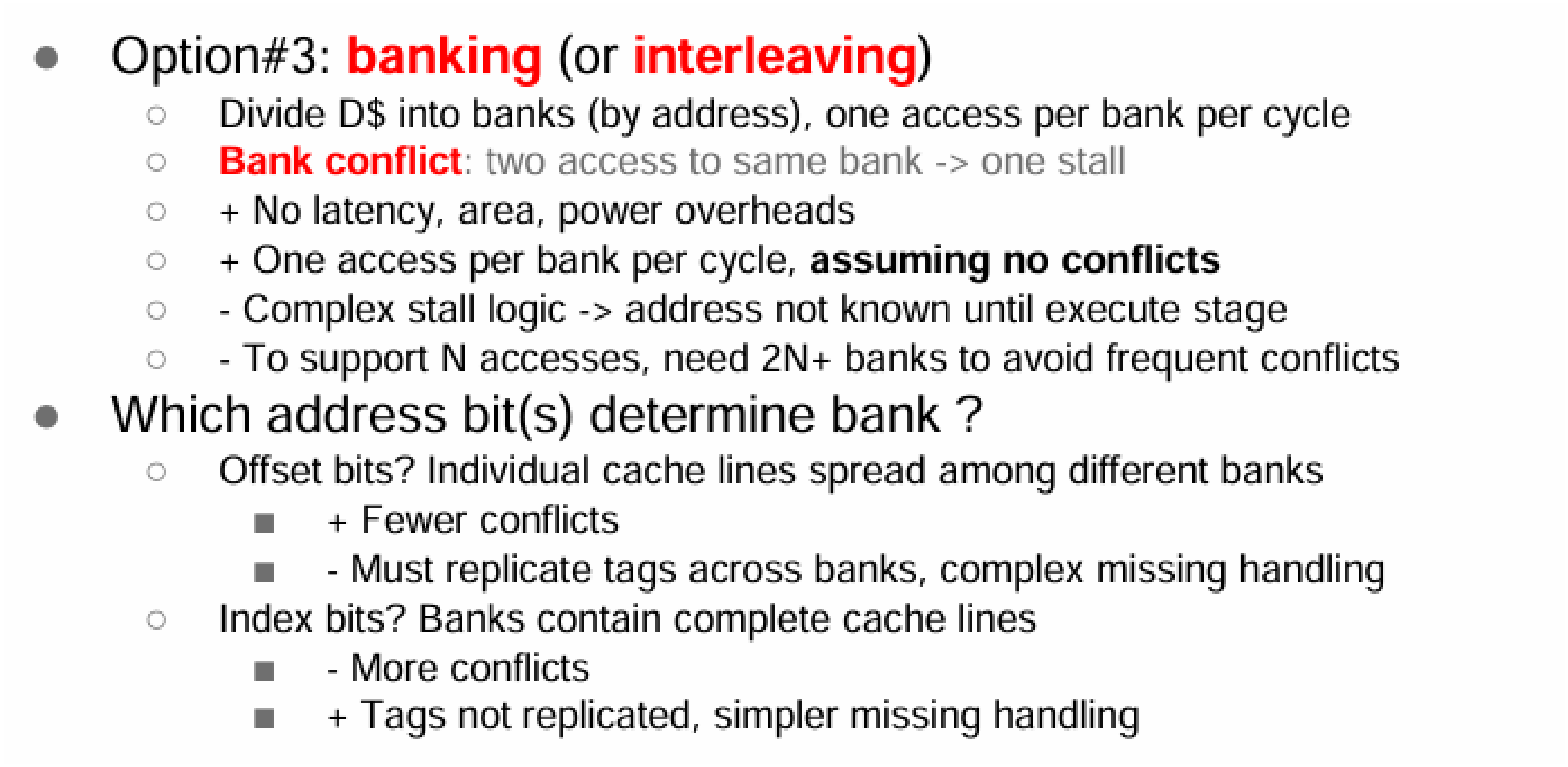
what is involved in accessing memory for multiple instructions per cycle?

- multi-banked D$
  - requires bank assignment and conflict-detection logic
- (rough) instruction mix: 20% loads, 15% stores
  - for width N, we need about 0.2*N load ports, 0.15*N store ports

## Wide Memory Access

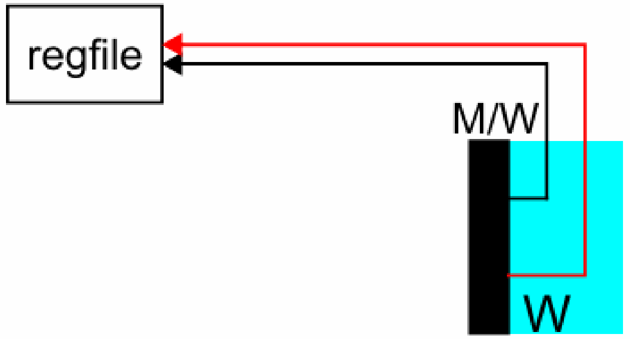
- How to provide additional D$ (D-cache) bandwidth ?
  - Have already seen split I$/D$, but that gives you just one D$ port
  - How to provide a second (maybe even a third) D$ port ?
- Option#1: **multi-porting**
  - + Most general solution, any two accesses per cycle
  - - Lots of wires; expansive in latency, area (cost), and power
- Option#2: **replication**
  - Read from either replica, but writes update both replicas
    - Writing both insures they have the same values
  - Multiplies read bandwidth only (writes must go to all replicas)
  - + General solution for loads, little latency penalty
  - - Not a solution for stores, area, power penalty

## Wide Memory Access



- Option#3: **banking** (or **interleaving**)
  - Divide D$ into banks (by address), one access per bank per cycle
  - **Bank conflict**: two access to same bank -> one stall
  - + No latency, area, power overheads
  - + One access per bank per cycle, **assuming no conflicts**
  - - Complex stall logic -> address not known until execute stage
  - - To support N accesses, need 2N+ banks to avoid frequent conflicts
- Which address bit(s) determine bank ?
  - Offset bits? Individual cache lines spread among different banks
    - + Fewer conflicts
    - - Must replicate tags across banks, complex missing handling
  - Index bits? Banks contain complete cache lines
    - - More conflicts
    - + Tags not replicated, simpler missing handling

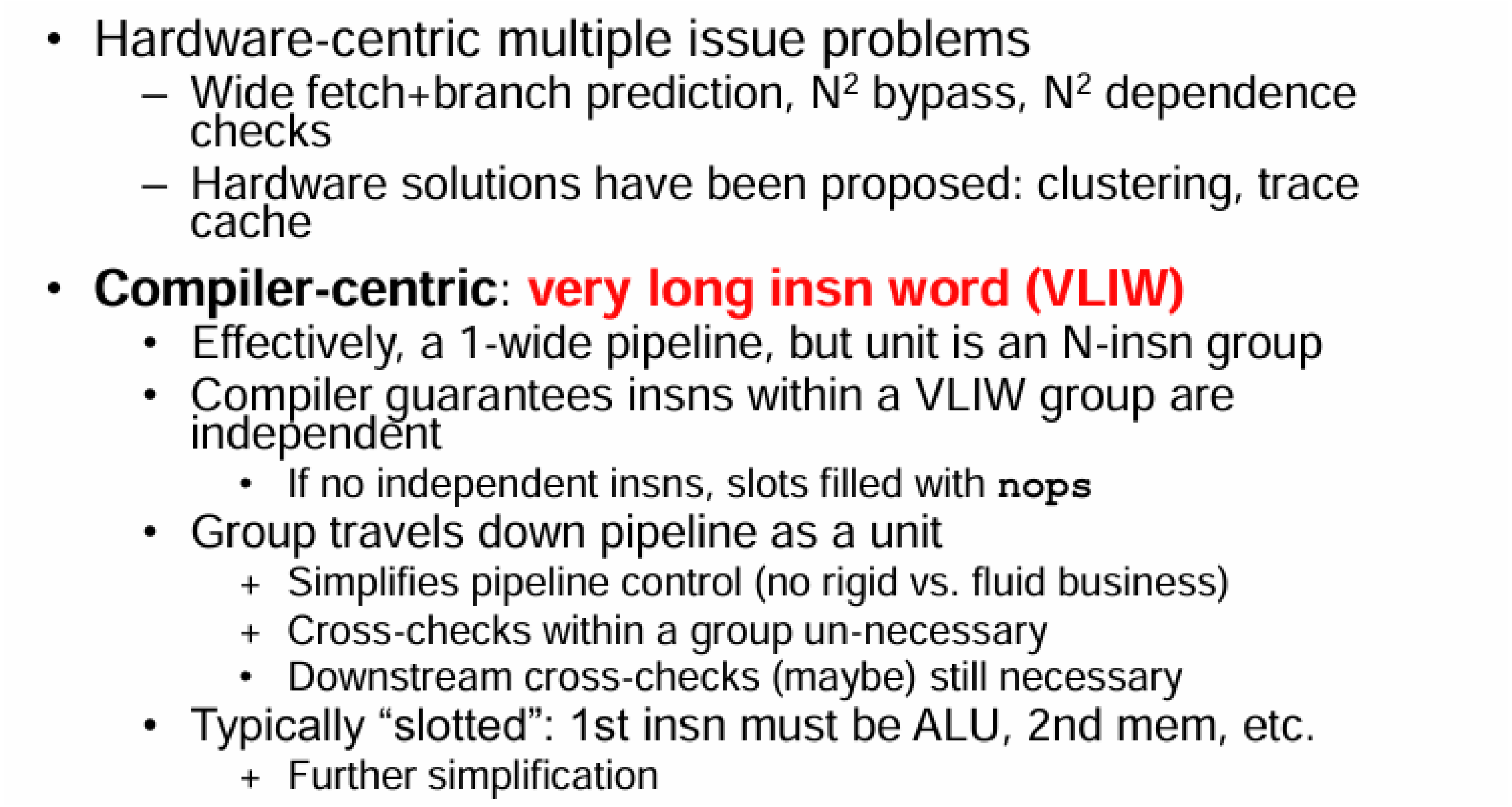
# Wide Writeback



what is involved in writing back multiple instructions per cycle?

- nothing too special, just another port on the register file
  - everything else is taken care of earlier in pipeline
- adding ports isn't free, though
  - increases area
  - increases access latency

## Very Long Instruction Word (VLIW)



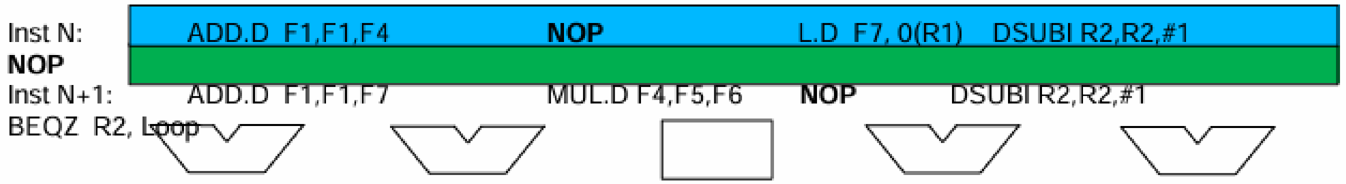
- Hardware-centric multiple issue problems
  - – Wide fetch+branch prediction, $N^2$ bypass, $N^2$ dependence checks
  - – Hardware solutions have been proposed: clustering, trace cache
- **Compiler-centric**: **very long insn word (VLIW)**
  - Effectively, a 1-wide pipeline, but unit is an N-insn group
  - Compiler guarantees insns within a VLIW group are independent
    - If no independent insns, slots filled with **`nops`**
  - Group travels down pipeline as a unit
    - + Simplifies pipeline control (no rigid vs. fluid business)
    - + Cross-checks within a group un-necessary
    - Downstream cross-checks (maybe) still necessary
  - Typically “slotted”: 1st insn must be ALU, 2nd mem, etc.
    - + Further simplification

## Very Long Instruction Word (VLIW)

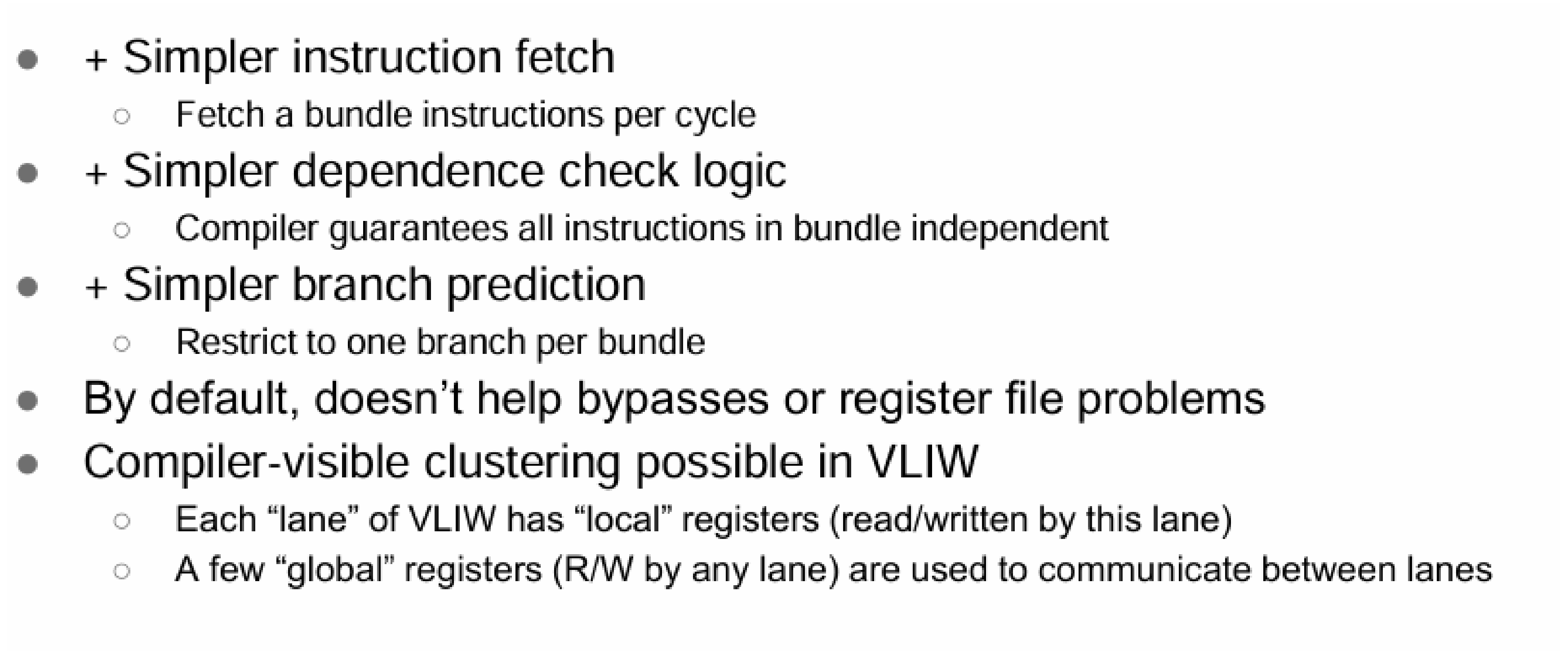
**VLIW**: instructions that "encode" multiple operations.

The hardware executes the entire instruction "at once" on parallel function units in execute stage.

The "long instructions" encode the fact that the operations are independent. So, the hardware does not need to dynamically figure this out. This can save area and power and is now popular in embedded processors (e.g., Texas Instruments C6x series of DSPs).

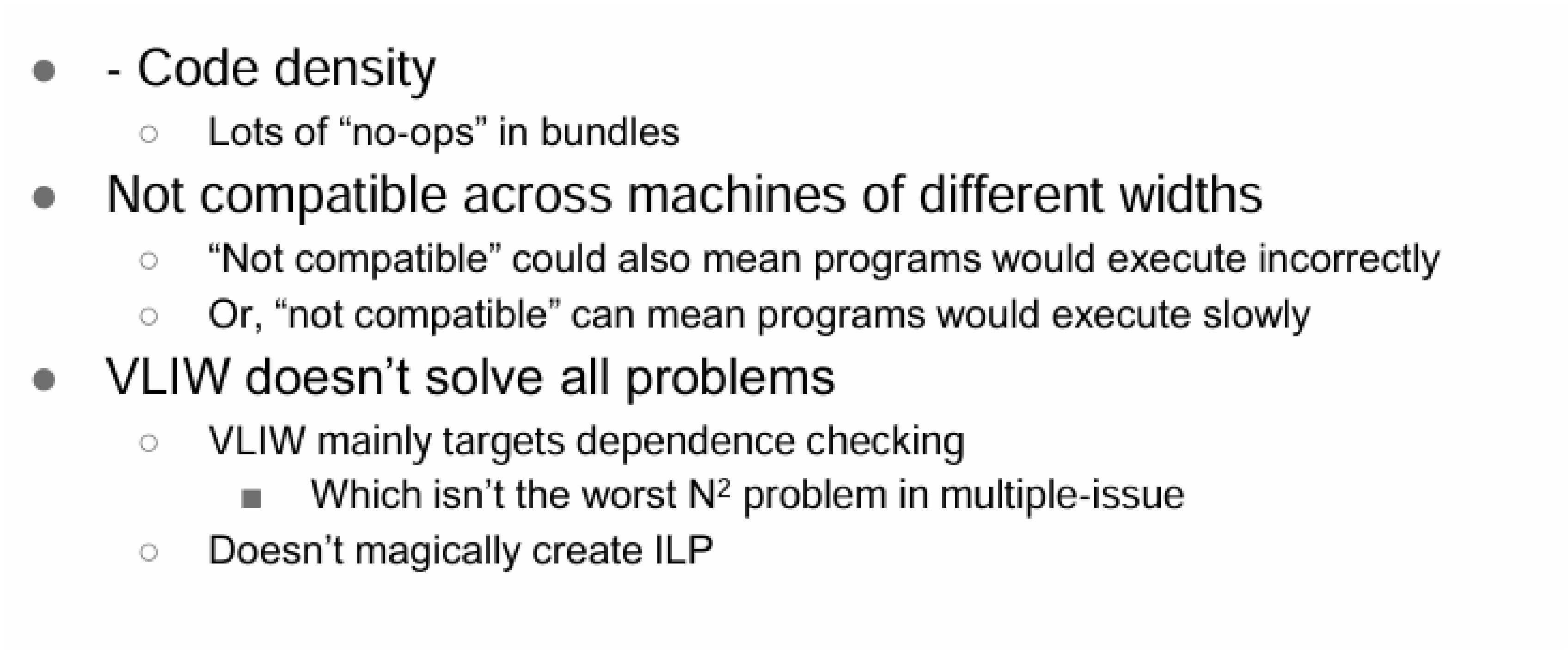


## Very Long Instruction Word (VLIW)



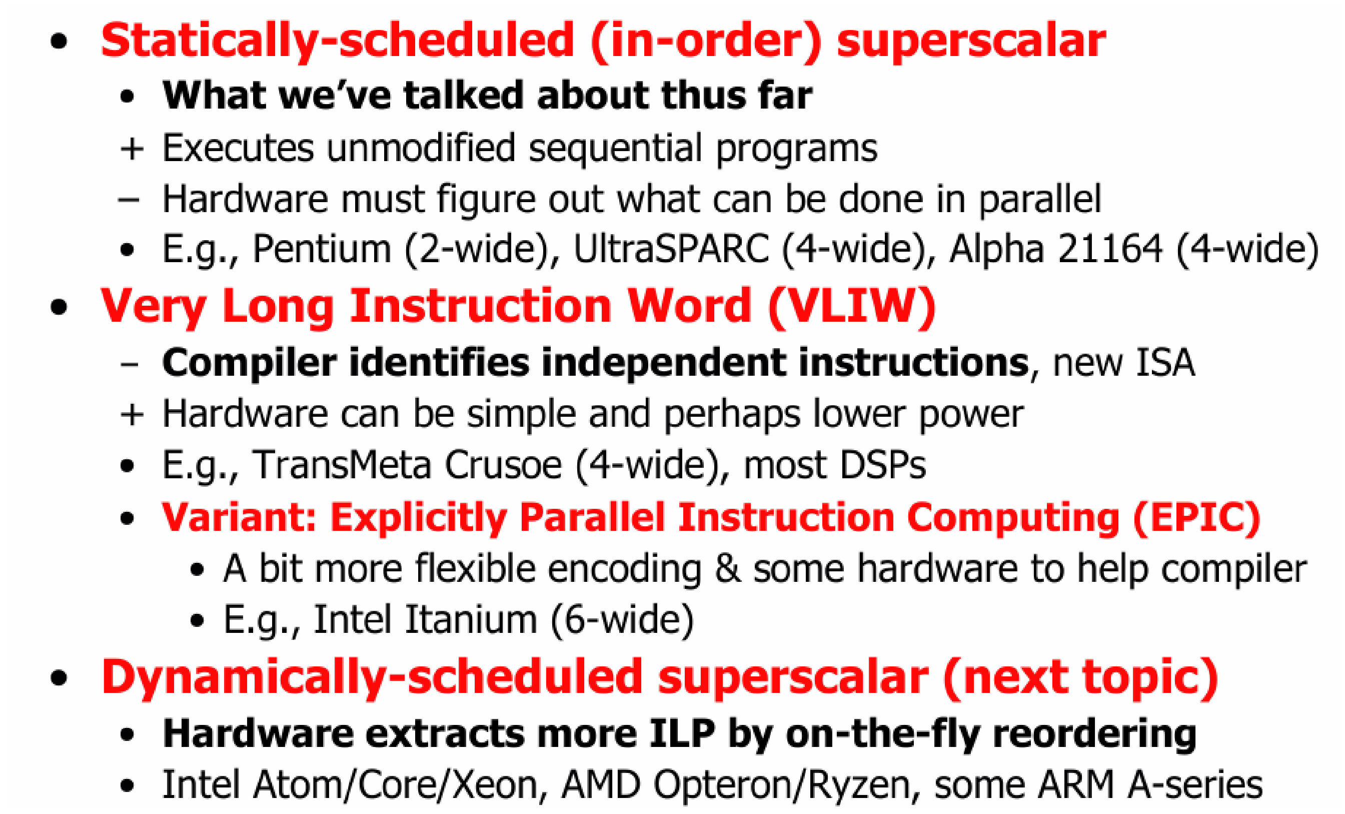
- + Simpler instruction fetch
  - Fetch a bundle instructions per cycle
- + Simpler dependence check logic
  - Compiler guarantees all instructions in bundle independent
- + Simpler branch prediction
  - Restrict to one branch per bundle
- By default, doesn’t help bypasses or register file problems
- Compiler-visible clustering possible in VLIW
  - Each “lane” of VLIW has “local” registers (read/written by this lane)
  - A few “global” registers (R/W by any lane) are used to communicate between lanes

## Very Long Instruction Word (VLIW)



- - Code density
  - Lots of “no-ops” in bundles
- Not compatible across machines of different widths
  - “Not compatible” could also mean programs would execute incorrectly
  - Or, “not compatible” can mean programs would execute slowly
- VLIW doesn’t solve all problems
  - VLIW mainly targets dependence checking
    - Which isn’t the worst $N^2$ problem in multiple-issue
  - Doesn’t magically create ILP

## Multiple-Issue Implementations



- **Statically-scheduled (in-order) superscalar**
  - **What we've talked about thus far**
  - + Executes unmodified sequential programs
  - – Hardware must figure out what can be done in parallel
  - E.g., Pentium (2-wide), UltraSPARC (4-wide), Alpha 21164 (4-wide)
- **Very Long Instruction Word (VLIW)**
  - – **Compiler identifies independent instructions**, new ISA
  - + Hardware can be simple and perhaps lower power
  - E.g., TransMeta Crusoe (4-wide), most DSPs
  - **Variant: Explicitly Parallel Instruction Computing (EPIC)**
    - A bit more flexible encoding & some hardware to help compiler
    - E.g., Intel Itanium (6-wide)
- **Dynamically-scheduled superscalar (next topic)**
  - **Hardware extracts more ILP by on-the-fly reordering**
  - Intel Atom/Core/Xeon, AMD Opteron/Ryzen, some ARM A-series

## Trends in Single-Processor Multiple Issue

| | 486 | Pentium | PentiumII | Pentium4 | Itanium | ItaniumII | Core2 |
|---|---|---|---|---|---|---|---|
| Year | 1989 | 1993 | 1998 | 2001 | 2002 | 2004 | 2006 |
| Width | 1 | 2 | 3 | 3 | 3 | 6 | 4 |

- Issue width has saturated at 4-6 for high-performance cores
  - Canceled Alpha 21464 was 8-way issue
  - Not enough ILP to justify going to wider issue
  - Hardware or compiler *scheduling* needed to exploit 4-6 effectively
    - More on this in the next unit

- For high-performance ***per watt*** cores (say, smart phones)
  - Typically 2-wide superscalar (but increasing each generation)

## Multiple Issue Summary

- Multiple issue
  - Exploits insn level parallelism (ILP) beyond pipelining
  - Improves IPC, but perhaps at some clock & energy penalty
  - 4-6 way issue is about the peak issue width currently justifiable
    - Low-power implementations today typically 2-wide superscalar

- Problem spots
  - $N^2$ bypass & register file $\rightarrow$ clustering
  - Fetch + branch prediction $\rightarrow$ buffering, loop streaming, trace cache
  - $N^2$ dependency check $\rightarrow$ VLIW/EPIC (but unclear how key this is)

- Implementations
  - Superscalar vs. VLIW/EPIC

## Out-of-Order Processor

- **Dynamically-scheduled processors**
  - Also called **out-of-order** processors
  - Hardware re-schedules insns…
  - …within a sliding window of VonNeumann insns
  - As with pipelining and superscalar, ISA unchanged
    - Same hardware/software interface, illusion of in-order
- Increases scheduling scope
  - Does loop unrolling transparently!
  - Uses branch prediction to “unroll” branches
- Examples:
  - first appeared in Pentium Pro (1995)
  - part of every smartphone/tablet/laptop/desktop/server chip

# In-Order Limitations

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ld [r1] → r2 | F | D | X | $M_1$ | $M_2$ | W | | | | | | | |
| add r2 + r3 → r4 | F | D | d* | d* | d* | X | $M_1$ | $M_2$ | W | | | | |
| xor r4 ^ r5 → r6 | | F | D | d* | d* | d* | X | $M_1$ | $M_2$ | W | | | |
| ld [r7] → r4 | | F | D | p* | p* | p* | X | $M_1$ | $M_2$ | W | | | |

- In-order pipeline, three-cycle load-use penalty
  - 2-wide
- Why not the following?

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ld [r1] → r2 | F | D | X | $M_1$ | $M_2$ | W | | | | | | | |
| add r2 + r3 → **r4** | F | D | d* | d* | d* | X | $M_1$ | $M_2$ | W | | | | |
| xor **r4** ^ r5 → r6 | | F | D | d* | d* | d* | X | $M_1$ | $M_2$ | W | | | |
| ld [r7] → **r4** | | F | D | **X** | **$M_1$** | **$M_2$** | **W** | | | | | | |

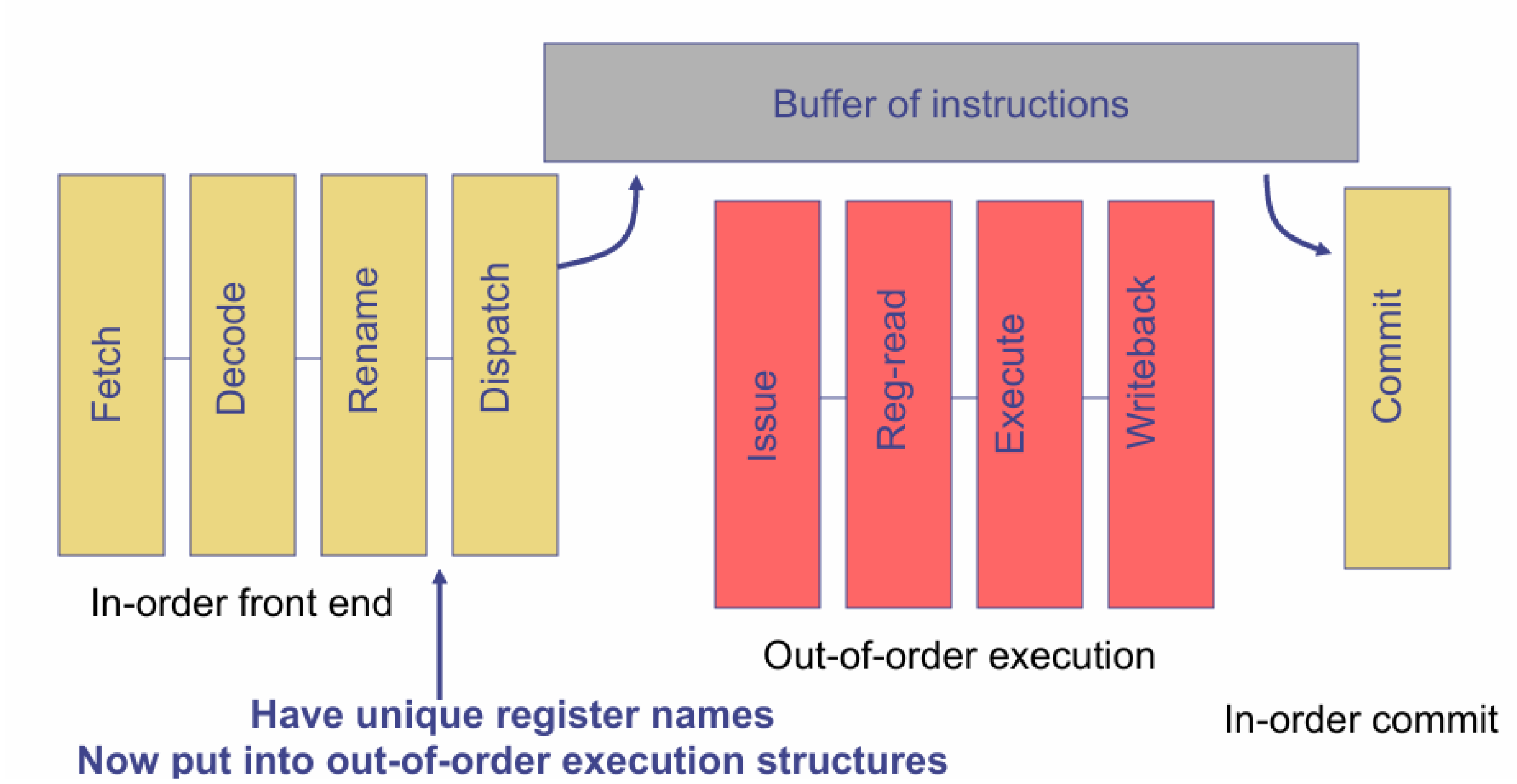
## In-Order Limitations

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ld [p1] → p2 | F | D | X | $M_1$ | $M_2$ | W | | | | | | | |
| add p2 + p3 → p4 | F | D | **d*** | **d*** | **d*** | X | $M_1$ | $M_2$ | W | | | | |
| xor p4 ^ p5 → p6 | | F | D | **d*** | **d*** | **d*** | X | $M_1$ | $M_2$ | W | | | |
| ld [p7] → p8 | | F | D | **p*** | **p*** | **p*** | X | $M_1$ | $M_2$ | W | | | |

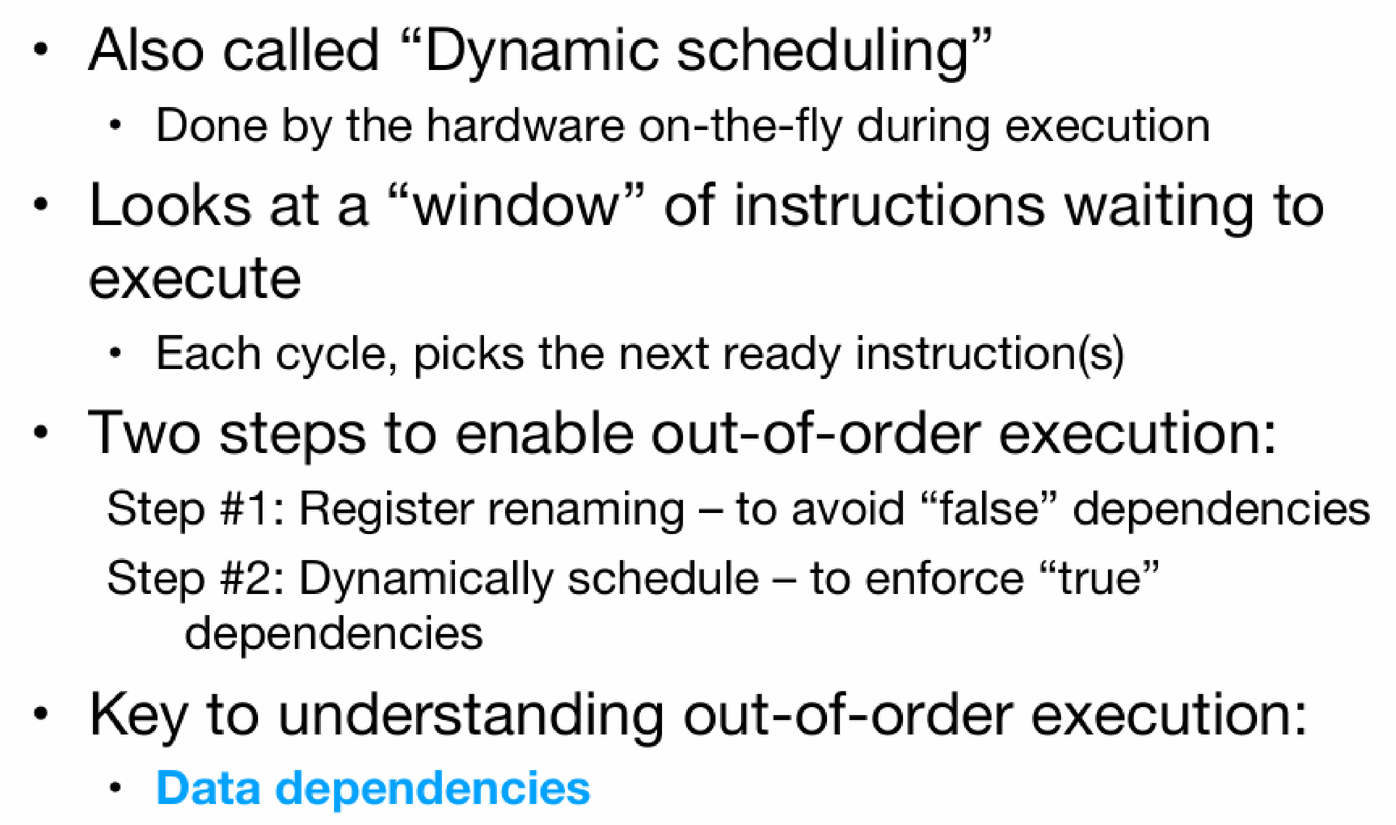
- In-order pipeline, three-cycle load-use penalty
  - 2-wide
- Why not the following:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ld [p1] → p2 | F | D | X | $M_1$ | $M_2$ | W | | | | | | | |
| add p2 + p3 → **p4** | F | D | **d*** | **d*** | **d*** | X | $M_1$ | $M_2$ | W | | | | |
| xor **p4** ^ p5 → p6 | | F | D | **d*** | **d*** | **d*** | X | $M_1$ | $M_2$ | W | | | |
| ld [p7] → **p8** | | F | D | **X** | $\mathbf{M_1}$ | $\mathbf{M_2}$ | **W** | | | | | | |

## Out-of-Order Pipeline



## Out-of-Order Execution

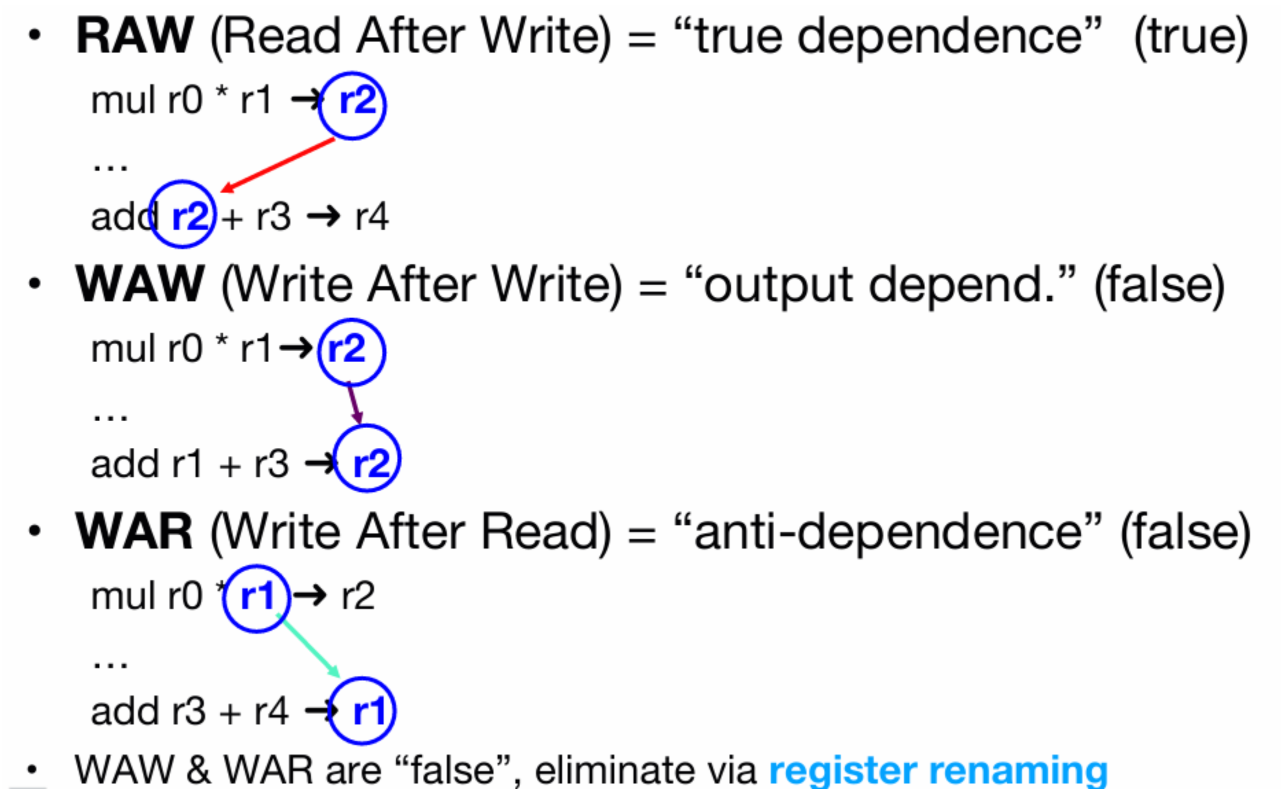


- Also called “Dynamic scheduling”
  - Done by the hardware on-the-fly during execution
- Looks at a “window” of instructions waiting to execute
  - Each cycle, picks the next ready instruction(s)
- Two steps to enable out-of-order execution:

  Step #1: Register renaming – to avoid “false” dependencies

  Step #2: Dynamically schedule – to enforce “true” dependencies
- Key to understanding out-of-order execution:
  - **Data dependencies**

## Dependence Types



- **RAW** (Read After Write) = “true dependence” (true)

  mul r0 * r1 → **r2**
  …
  add **r2** + r3 → r4

- **WAW** (Write After Write) = “output depend.” (false)

  mul r0 * r1→**r2**
  …
  add r1 + r3 → **r2**

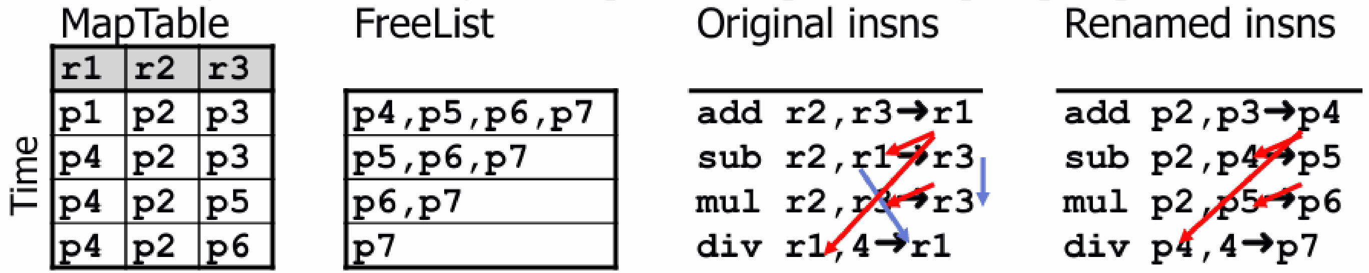
- **WAR** (Write After Read) = “anti-dependence” (false)

  mul r0 * **r1** → r2
  …
  add r3 + r4 → **r1**

- WAW & WAR are “false”, eliminate via **register renaming**

## Register Renaming

- To eliminate register conflicts/hazards
- "Architected" vs "Physical" registers – level of indirection
  - Names: **r1,r2,r3**
  - Locations: **p1,p2,p3,p4,p5,p6,p7**
  - Original mapping: **r1**→**p1**, **r2**→**p2**, **r3**→**p3**, **p4**–**p7** are free



MapTable (Time ↓)

| r1 | r2 | r3 |
|---|---|---|
| p1 | p2 | p3 |
| p4 | p2 | p3 |
| p4 | p2 | p5 |
| p4 | p2 | p6 |

FreeList

| |
|---|
| p4,p5,p6,p7 |
| p5,p6,p7 |
| p6,p7 |
| p7 |

Original insns

```
add r2,r3➔r1
sub r2,r1➔r3
mul r2,r3➔r3
div r1,4➔r1
```

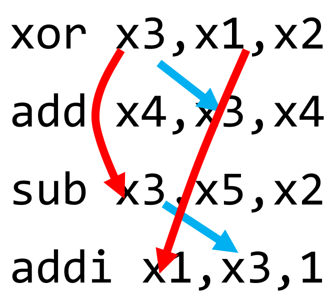
Renamed insns

```
add p2,p3➔p4
sub p2,p4➔p5
mul p2,p5➔p6
div p4,4➔p7
```

- Renaming – conceptually write each register once
  - + Removes **false** dependences
  - + Leaves **true** dependences intact!

# Register Renaming Example

true/false
dependencies



| original insns | renamed insns |
|---|---|
| `xor x3,x1,x2` | |
| `add x4,x3,x4` | |
| `sub x3,x5,x2` | |
| `addi x1,x3,1` | |

| arch reg | phys reg |
|---|---|
| x1 | ~~p1~~ p9 |
| x2 | p2 |
| x3 | ~~p3~~ ~~p6~~ p8 |
| x4 | ~~p4~~ p7 |
| x5 | p5 |

| free list |
|---|
| ~~p6~~ |
| ~~p7~~ |
| ~~p8~~ |
| ~~p9~~ |
| p10 |

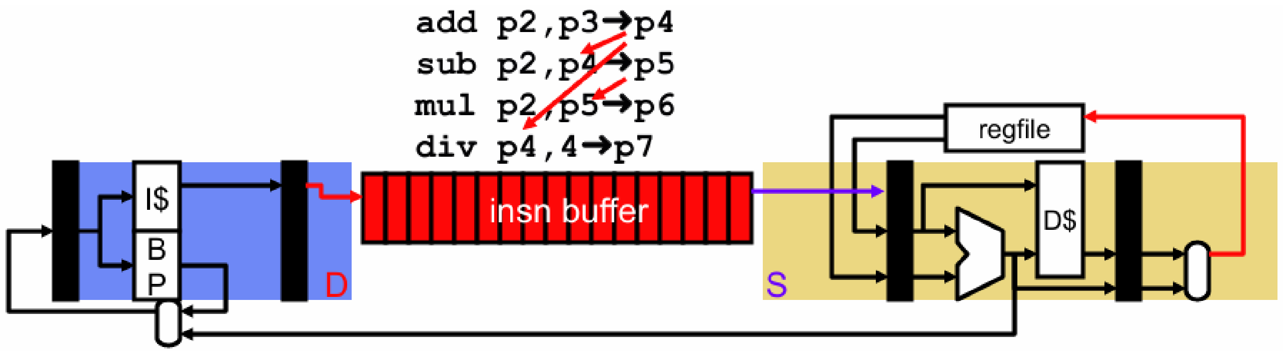
## Register Renaming Algorithm

- Two key data structures:
  - maptable[architectural_reg] ➔ physical_reg
  - Free list: allocate & free registers
- Algorithm: at "decode" stage for each instruction:

```
insn.phys_input1 = maptable[insn.arch_input1]
insn.phys_input2 = maptable[insn.arch_input2]
insn.old_phys_output = maptable[insn.arch_output]
new_reg = new_phys_reg()
maptable[insn.arch_output] = new_reg
insn.phys_output = new_reg
```

- **At "commit"**
  - **Once all older instructions have committed, free register**
    **`free_phys_reg(insn.old_phys_output)`**

# Dynamic Scheduling Overview



Ready Table

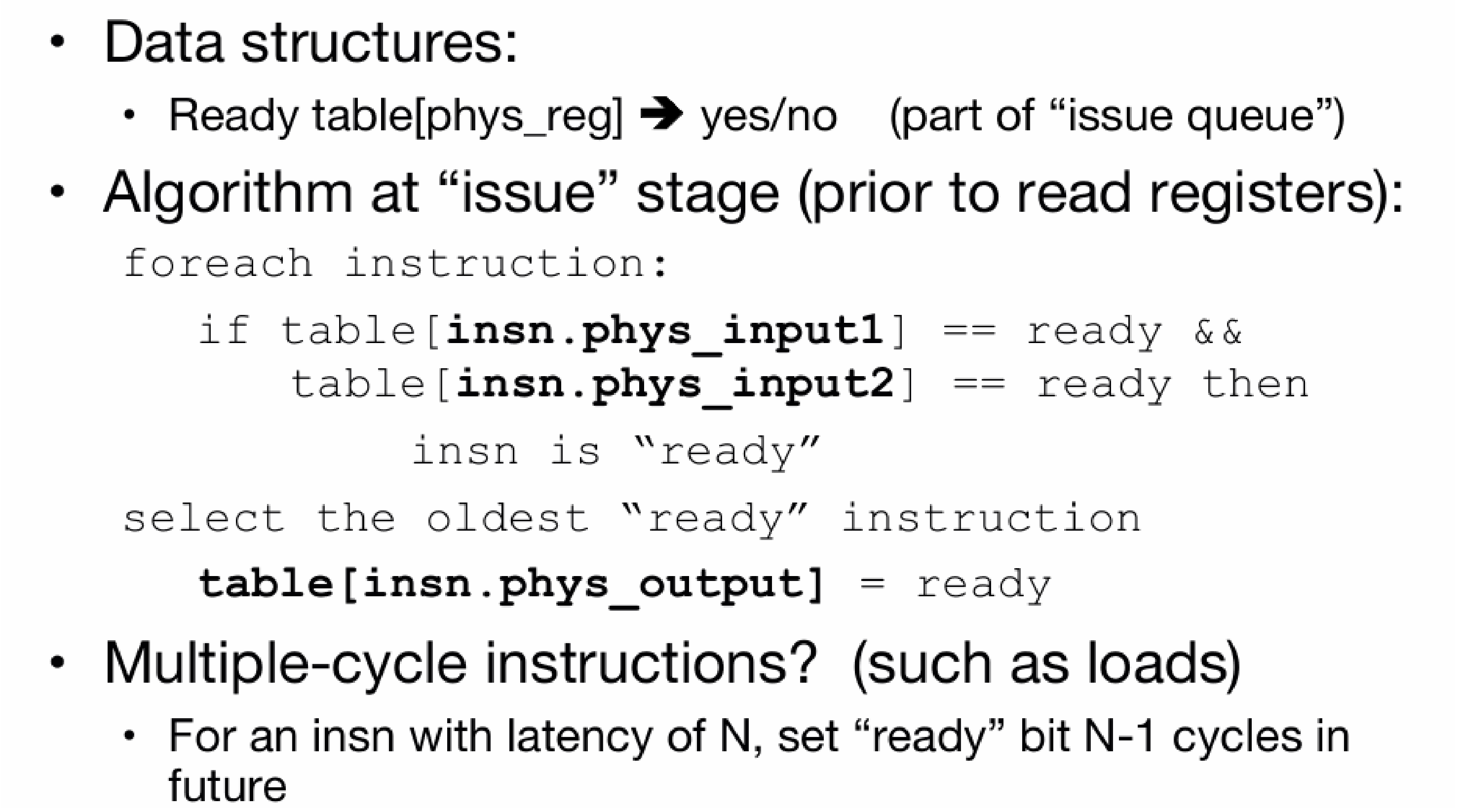
Time

| P2 | P3 | P4 | P5 | P6 | P7 |
|---|---|---|---|---|---|
| Yes | Yes | | | | |
| Yes | Yes | Yes | | | |
| Yes | Yes | Yes | Yes | | Yes |
| Yes | Yes | Yes | Yes | Yes | Yes |

```
add p2,p3→p4
sub p2,p4→p5 and   div p4,4→p7
mul p2,p5→p6
```

- Insns fetch/decode/rename into *Instruction Buffer*
  - Also called "instruction window" or "instruction scheduler"
- Insns (conceptually) check ready bits each cycle
  - Execute oldest "ready" instruction, set output as "ready"

## Dynamic Scheduling Algorithm



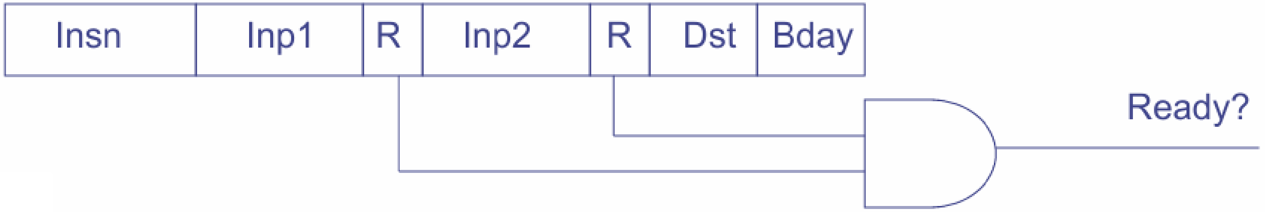
- Data structures:
  - Ready table[phys_reg] ➔ yes/no (part of "issue queue")
- Algorithm at "issue" stage (prior to read registers):

```
foreach instruction:
    if table[insn.phys_input1] == ready &&
         table[insn.phys_input2] == ready then
             insn is "ready"
select the oldest "ready" instruction
    table[insn.phys_output] = ready
```

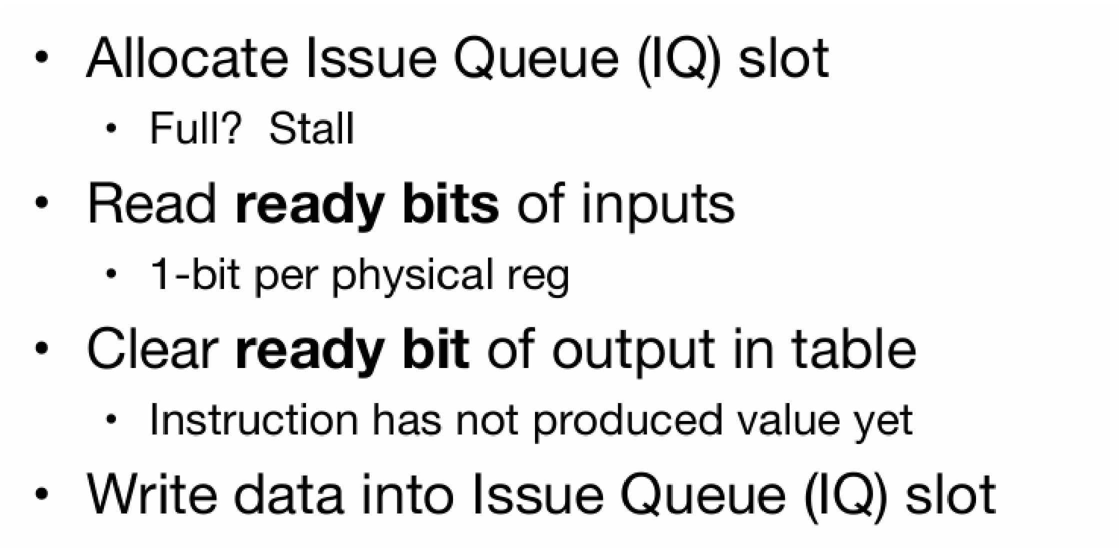
- Multiple-cycle instructions? (such as loads)
  - For an insn with latency of N, set "ready" bit N-1 cycles in future

## Dispatch

- Put renamed instructions into OoO structures
- Re-order buffer (ROB)
  - Holds instructions from Fetch through Commit
- Issue Queue
  - Central piece of scheduling logic
  - Holds instructions from Dispatch through Issue
  - Tracks ready inputs
    - Physical register names + ready bit
    - "AND" the bits to tell if ready

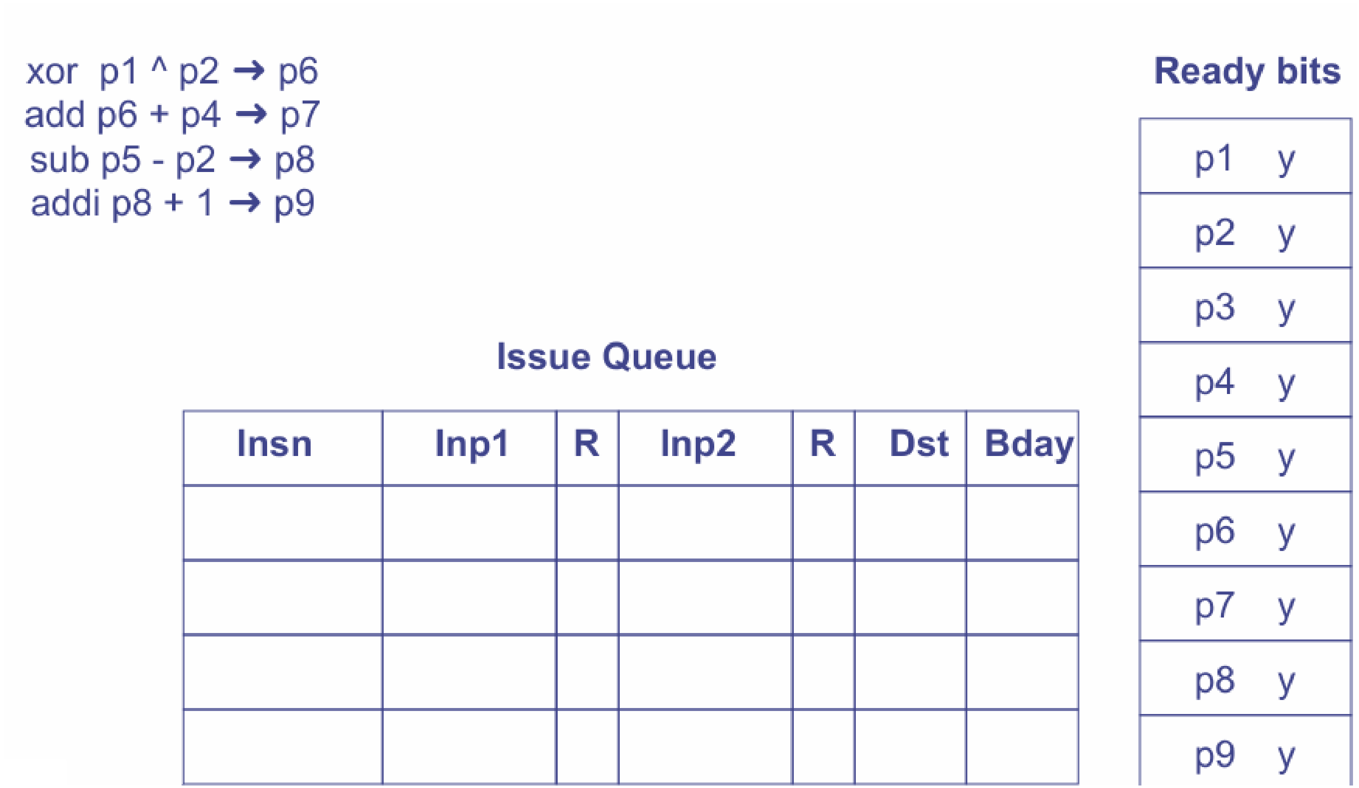


## Dispatch Steps



- Allocate Issue Queue (IQ) slot
  - Full? Stall
- Read **ready bits** of inputs
  - 1-bit per physical reg
- Clear **ready bit** of output in table
  - Instruction has not produced value yet
- Write data into Issue Queue (IQ) slot

## Dispatch Example



```
xor  p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8
addi p8 + 1 → p9
```

**Issue Queue**

| Insn | Inp1 | R | Inp2 | R | Dst | Bday |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

**Ready bits**

| | |
|---|---|
| p1 | y |
| p2 | y |
| p3 | y |
| p4 | y |
| p5 | y |
| p6 | y |
| p7 | y |
| p8 | y |
| p9 | y |

## Dispatch Example

xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8
addi p8 + 1 → p9

**Issue Queue**

| Insn | Inp1 | R | Inp2 | R | Dst | Bday |
|---|---|---|---|---|---|---|
| xor | p1 | y | p2 | y | p6 | 0 |
| | | | | | | |
| | | | | | | |
| | | | | | | |

**Ready bits**

| | |
|---|---|
| p1 | y |
| p2 | y |
| p3 | y |
| p4 | y |
| p5 | y |
| **p6** | **n** |
| p7 | y |
| p8 | y |
| p9 | y |

# Dispatch Example

xor p1 ^ p2 ➔ p6
add p6 + p4 ➔ p7
sub p5 - p2 ➔ p8
addi p8 + 1 ➔ p9

**Issue Queue**

| Insn | Inp1 | R | Inp2 | R | Dst | Bday |
|---|---|---|---|---|---|---|
| xor | p1 | y | p2 | y | p6 | 0 |
| add | p6 | n | p4 | y | p7 | 1 |
| | | | | | | |
| | | | | | | |

**Ready bits**

| | |
|---|---|
| p1 | y |
| p2 | y |
| p3 | y |
| p4 | y |
| p5 | y |
| p6 | n |
| **p7** | **n** |
| p8 | y |
| p9 | y |

## Dispatch Example

xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8
addi p8 + 1 → p9

**Issue Queue**

| Insn | Inp1 | R | Inp2 | R | Dst | Bday |
|---|---|---|---|---|---|---|
| xor | p1 | y | p2 | y | p6 | 0 |
| add | p6 | n | p4 | y | p7 | 1 |
| sub | p5 | y | p2 | y | p8 | 2 |
| | | | | | | |

**Ready bits**

| | |
|---|---|
| p1 | y |
| p2 | y |
| p3 | y |
| p4 | y |
| p5 | y |
| p6 | n |
| p7 | n |
| **p8** | **n** |
| p9 | y |

## Dispatch Example

xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8
addi p8 + 1 → p9

**Issue Queue**

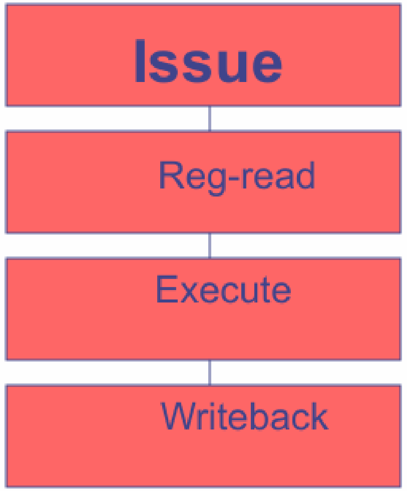
| Insn | Inp1 | R | Inp2 | R | Dst | Bday |
|---|---|---|---|---|---|---|
| xor | p1 | y | p2 | y | p6 | 0 |
| add | p6 | n | p4 | y | p7 | 1 |
| sub | p5 | y | p2 | y | p8 | 2 |
| addi | p8 | n | --- | y | p9 | 3 |

**Ready bits**

| | |
|---|---|
| p1 | y |
| p2 | y |
| p3 | y |
| p4 | y |
| p5 | y |
| p6 | n |
| p7 | n |
| p8 | n |
| **p9** | **n** |

# Out-of-order Pipeline

- Execution (out-of-order) stages
- **Select** ready instructions
  - Send for execution
- **Wakeup** dependents



# Issue = Select + Wakeup

- **Select** oldest of “ready” instructions
  - ➢ “xor” is the oldest ready instruction below
  - ➢ “xor” and “sub” are the two oldest ready instructions below
  - May have resource constraints, e.g., can’t do 2 loads

| Insn | Inp1 | R | Inp2 | R | Dst | Bday | |
|---|---|---|---|---|---|---|---|
| xor | p1 | **y** | p2 | **y** | p6 | 0 | **Ready!** |
| add | p6 | n | p4 | y | p7 | 1 | |
| sub | p5 | **y** | p2 | **y** | p8 | 2 | **Ready!** |
| addi | p8 | n | --- | y | p9 | 3 | |

## Issue = Select + Wakeup

- Wakeup dependent instructions
  - Search for destination (Dst) in inputs & set "ready" bit
    - Implemented with a special memory array circuit called a Content Addressable Memory (CAM)
  - Also update ready-bit table for future instructions

| Insn | Inp1 | R | Inp2 | R | Dst | Bday |
|---|---|---|---|---|---|---|
| xor | p1 | y | p2 | y | **p6** | 0 |
| add | **p6** | **y** | p4 | y | p7 | 1 |
| sub | p5 | y | p2 | y | **p8** | 2 |
| addi | **p8** | **y** | --- | y | p9 | 3 |

Ready bits

| | |
|---|---|
| p1 | y |
| p2 | y |
| p3 | y |
| p4 | y |
| p5 | y |
| **p6** | **y** |
| p7 | n |
| **p8** | **y** |
| p9 | n |

  - For multi-cycle operations (loads, floating point)
    - Wakeup deferred a few cycles
    - Include checks to avoid structural hazards

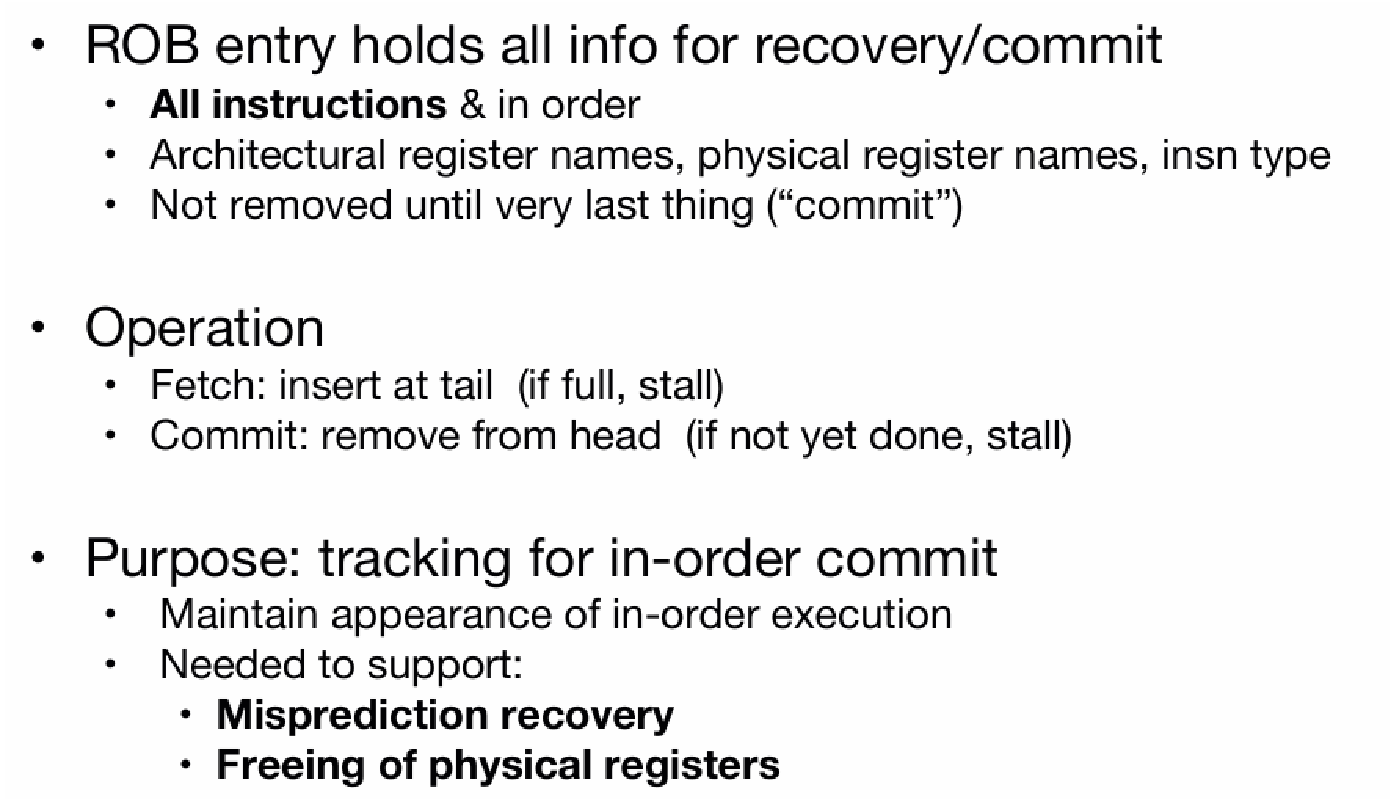
## Issue

- **Select/Wakeup** one cycle
- Dependent instructions execute on back-to-back cycles
  - Next cycle: add/addi are ready:

| Insn | Inp1 | R | Inp2 | R | Dst | Bday |
|---|---|---|---|---|---|---|
| | | | | | | |
| add | p6 | **y** | p4 | y | p7 | 1 |
| | | | | | | |
| addi | p8 | **y** | --- | y | p9 | 3 |

- Issued instructions are removed from issue queue
  - Free up space for subsequent instructions

# Re-order Buffer (ROB)

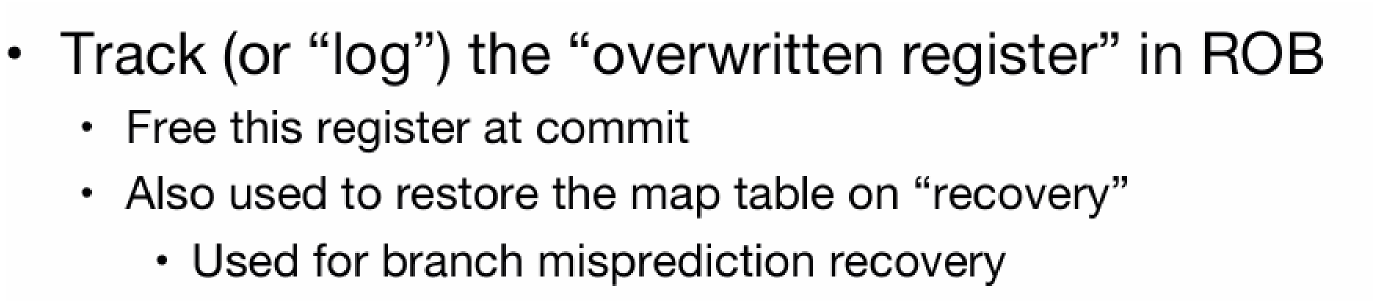


- ROB entry holds all info for recovery/commit
  - **All instructions** & in order
  - Architectural register names, physical register names, insn type
  - Not removed until very last thing (“commit”)

- Operation
  - Fetch: insert at tail  (if full, stall)
  - Commit: remove from head  (if not yet done, stall)

- Purpose: tracking for in-order commit
  - Maintain appearance of in-order execution
  - Needed to support:
    - **Misprediction recovery**
    - **Freeing of physical registers**

## Register Renaming Revisited



- Track (or “log”) the “overwritten register” in ROB
  - Free this register at commit
  - Also used to restore the map table on “recovery”
    - Used for branch misprediction recovery

## Recovery

- Completely remove wrong path instructions
  - Flush from IQ
  - Remove from ROB
  - Restore map table to before misprediction
  - Free destination registers
- How to restore map table?
  - Option #1: log-based reverse renaming (on following slides)
    - Tracks the old mapping to allow it to be reversed
    - Done sequentially for each instruction (slow)
  - Option #2: checkpoint-based recovery
    - Checkpoint state of maptable and free list each cycle
    - Faster recovery, but requires more state
  - Option #3: hybrid (checkpoint branches, unwind for others)

# Reg Renaming Recovery Example

beq is midpredicted, but other insns have already been renamed. How do we restore map table & free list?

| original insns | renamed insns | overwritten |
|---|---|---|
| `beq ...` | `beq ...` | |
| ~~`xor x3,x1,x2`~~ | ~~`xor p6,p1,p2`~~ | ~~`x3:p3`~~ |
| ~~`add x4,x3,x4`~~ | ~~`add p7,p6,p4`~~ | ~~`x4:p4`~~ |
| ~~`sub x3,x5,x2`~~ | ~~`sub p8,p5,p2`~~ | ~~`x3:p6`~~ |
| ~~`addi x1,x3,1`~~ | ~~`addi p9,p8,1`~~ | ~~`x1:p1`~~ |

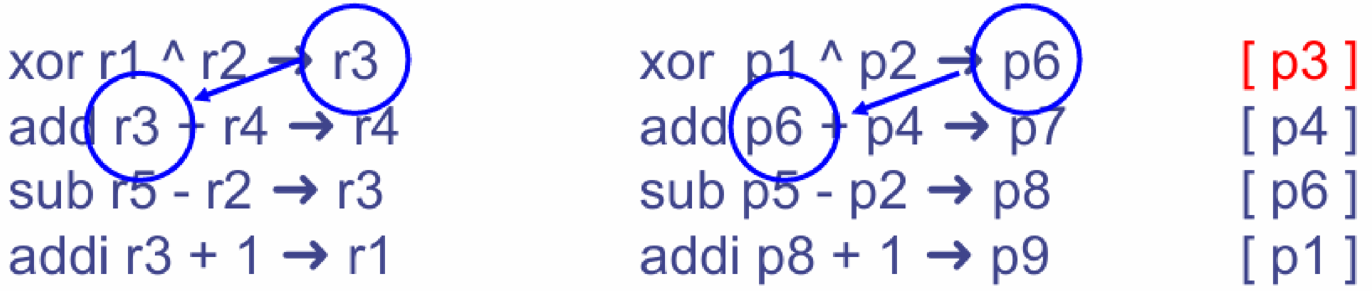
| arch reg | phys reg |
|---|---|
| x1 | ~~p9~~ p1 |
| x2 | p2 |
| x3 | ~~p8~~ ~~p6~~ p3 |
| x4 | ~~p7~~ p4 |
| x5 | p5 |

| free list |
|---|
| ~~p6~~ |
| ~~p7~~ |
| ~~p8~~ |
| ~~p9~~ |
| p10 |

## Commit

- At commit, an insn updates architected state
  - Commit is done in-order
  - Only when instructions are finished and there is no possibility of rollback
  - Ok to free overwritten register at this point

## Free over-written register



```
xor r1 ^ r2 → r3          xor  p1 ^ p2 → p6        [ p3 ]
add r3 + r4 → r4          add p6 + p4 → p7         [ p4 ]
sub r5 - r2 → r3          sub p5 - p2 → p8         [ p6 ]
addi r3 + 1 → r1          addi p8 + 1 → p9         [ p1 ]
```

- p3 was r3 before xor
- p6 is r3 after xor
  - Anything older than xor should read p3
  - Anything younger than xor should read p6 (until another insn writes r3)
  - At commit of xor, no older instructions exist

# Register Renaming Commit Example

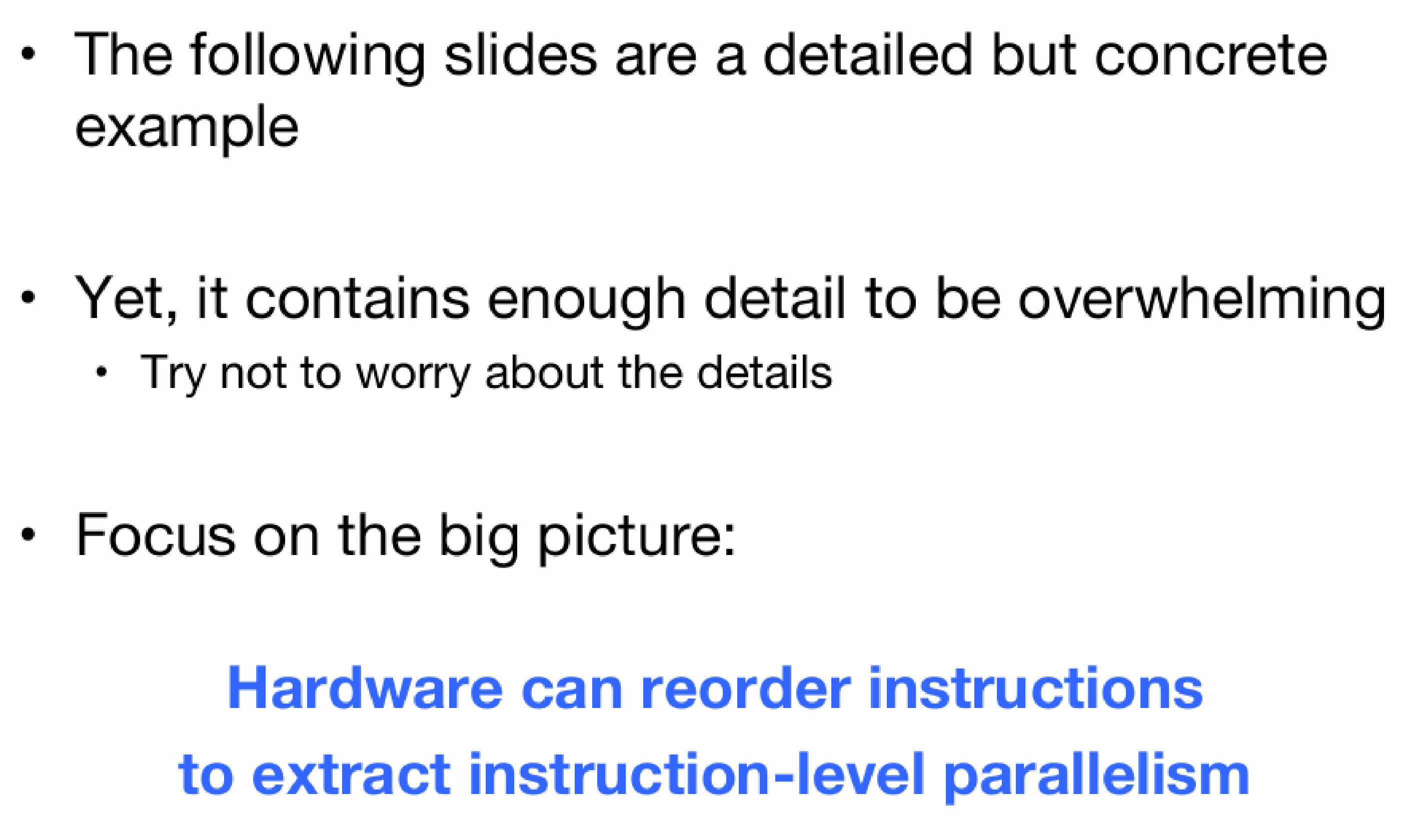
commit insns in-order, freeing phys registers

| original insns | renamed insns | overwritten |
|---|---|---|
| xor x3,x1,x2 | xor p6,p1,p2 | x3:p3 |
| add x4,x3,x4 | add p7,p6,p4 | x4:p4 |
| sub x3,x5,x2 | sub p8,p5,p2 | x3:p6 |
| addi x1,x3,1 | addi p9,p8,1 | x1:p1 |

| arch reg | phys reg |
|---|---|
| x1 | p9 |
| x2 | p2 |
| x3 | p8 |
| x4 | p7 |
| x5 | p5 |

| free list |
|---|
| p10 |
| |
| |
| |
| |

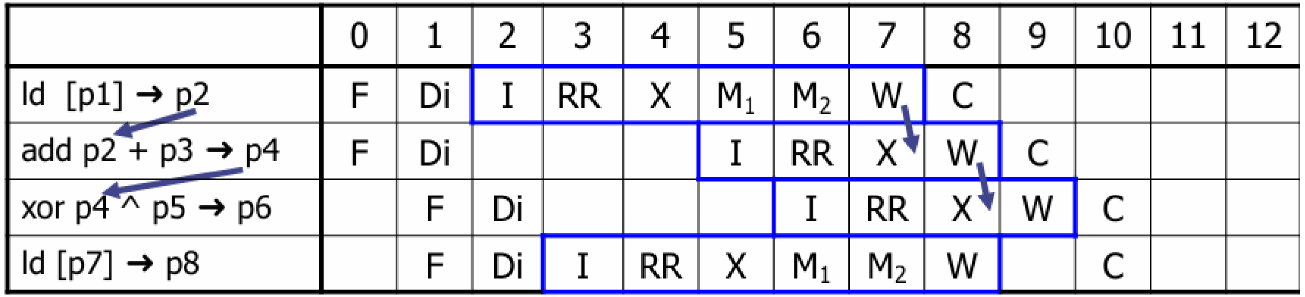
## Dynamic Scheduling Example



- The following slides are a detailed but concrete example
- Yet, it contains enough detail to be overwhelming
  - Try not to worry about the details
- Focus on the big picture:

**Hardware can reorder instructions to extract instruction-level parallelism**

# Dynamic Scheduling Example



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [p1] → p2 | F | Di | I | RR | X | $M_1$ | $M_2$ | W | C | | | | |
| add p2 + p3 → p4 | F | Di | | | | I | RR | X | W | C | | | |
| xor p4 ^ p5 → p6 | | F | Di | | | | I | RR | X | W | C | | |
| ld [p7] → p8 | | F | Di | I | RR | X | $M_1$ | $M_2$ | W | | C | | |

- How would this execution occur cycle-by-cycle?
- Execution latencies assumed in this example:
  - Loads have two-cycle load-to-use penalty
    - Three cycle total execution latency
  - All other instructions have single-cycle execution latency
- Issue queue holds all un-executed instructions
  - Holds ready/not-ready status
  - Faster than looking up in ready table each cycle

## Out-of-Order Pipeline – Cycle 0

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | | | | | | | | | | | | |
| add r2 + r3 → r4 | F | | | | | | | | | | | | |
| xor r4 ^ r5 → r6 | | | | | | | | | | | | | |
| ld [r7] → r4 | | | | | | | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p7 |
| r3 | p6 |
| r4 | p5 |
| r5 | p4 |
| r6 | p3 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | --- |
| p10 | --- |
| p11 | --- |
| p12 | --- |

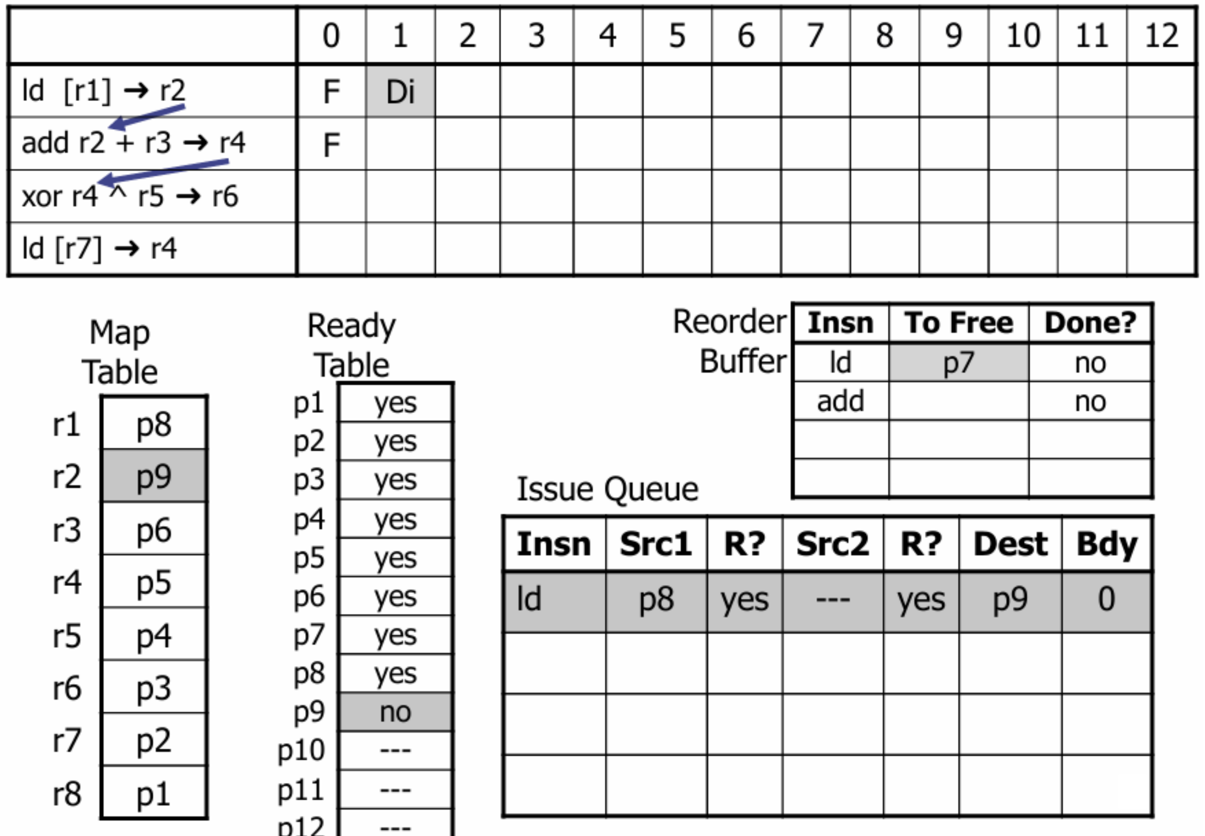
Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | | no |
| add | | no |
| | | |
| | | |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

# Out-of-Order Pipeline – Cycle 1a



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | | | | | | | | | | | |
| add r2 + r3 → r4 | F | | | | | | | | | | | | |
| xor r4 ^ r5 → r6 | | | | | | | | | | | | | |
| ld [r7] → r4 | | | | | | | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p5 |
| r5 | p4 |
| r6 | p3 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | no |
| p10 | --- |
| p11 | --- |
| p12 | --- |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | | no |
| | | |
| | | |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ld | p8 | yes | --- | yes | p9 | 0 |
| | | | | | | |
| | | | | | | |
| | | | | | | |

## Out-of-Order Pipeline – Cycle 1b

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | | | | | | | | | | | |
| add r2 + r3 → r4 | F | Di | | | | | | | | | | | |
| xor r4 ^ r5 → r6 | | | | | | | | | | | | | |
| ld [r7] → r4 | | | | | | | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p10 |
| r5 | p4 |
| r6 | p3 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | no |
| p10 | no |
| p11 | --- |
| p12 | --- |

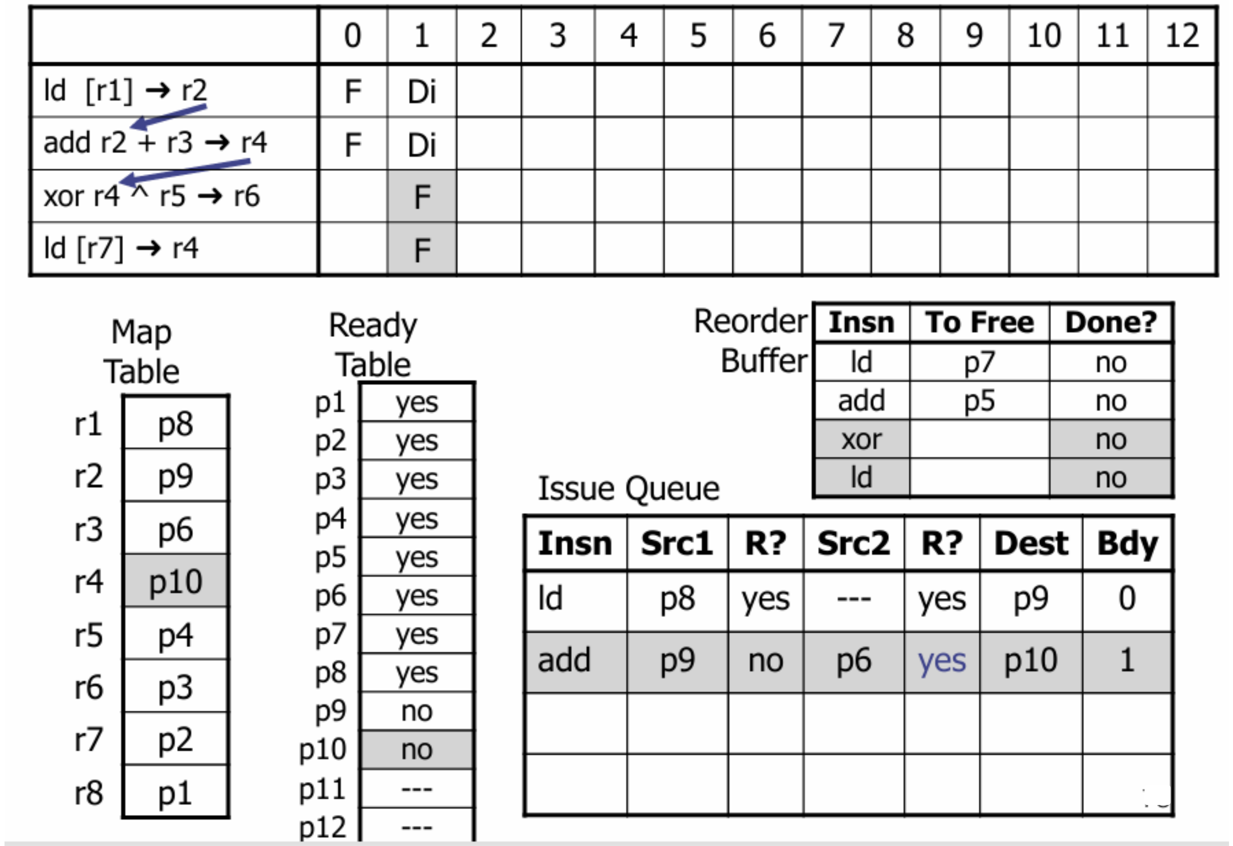
Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | p5 | no |
| | | |
| | | |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ld | p8 | yes | --- | yes | p9 | 0 |
| add | p9 | no | p6 | yes | p10 | 1 |
| | | | | | | |
| | | | | | | |

## Out-of-Order Pipeline – Cycle 1c



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | | | | | | | | | | | |
| add r2 + r3 → r4 | F | Di | | | | | | | | | | | |
| xor r4 ^ r5 → r6 | | F | | | | | | | | | | | |
| ld [r7] → r4 | | F | | | | | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p10 |
| r5 | p4 |
| r6 | p3 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | no |
| p10 | no |
| p11 | --- |
| p12 | --- |

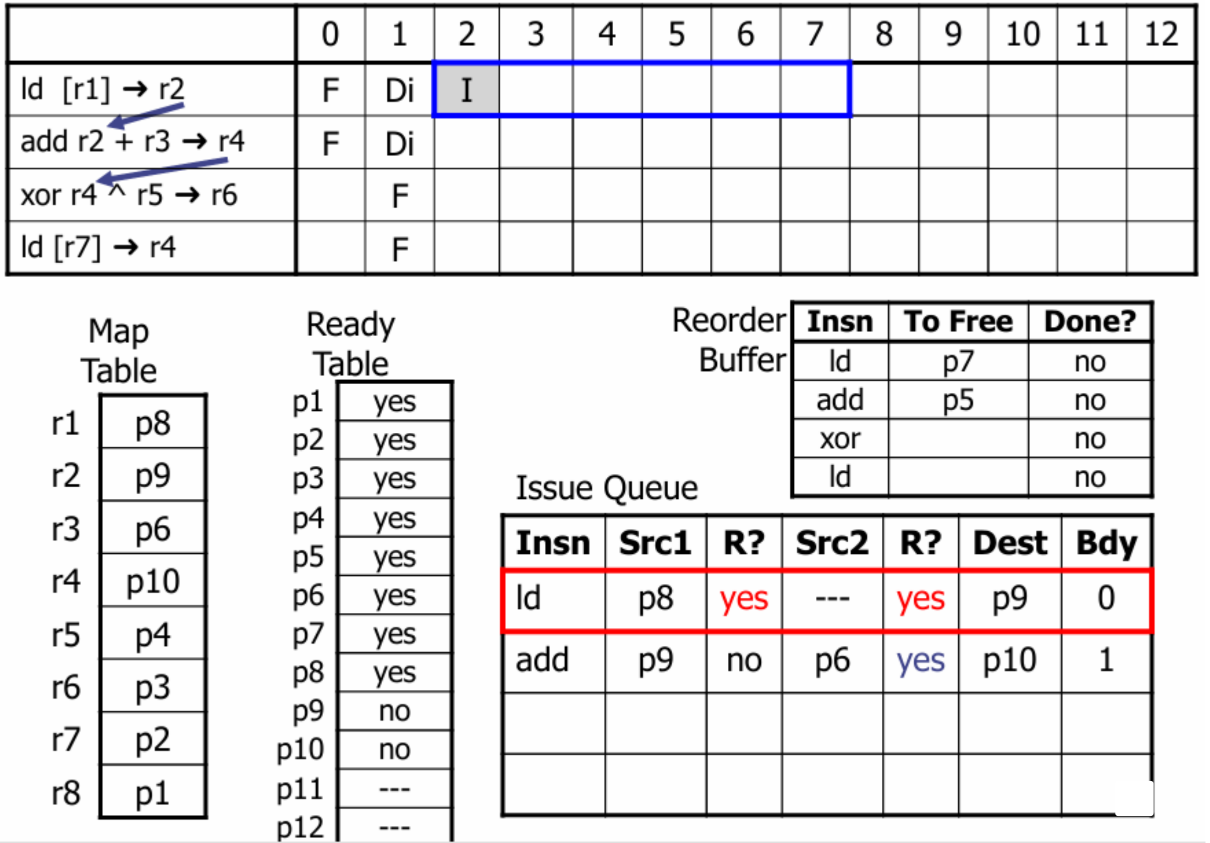
Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | p5 | no |
| xor | | no |
| ld | | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ld | p8 | yes | --- | yes | p9 | 0 |
| add | p9 | no | p6 | yes | p10 | 1 |
| | | | | | | |
| | | | | | | |

# Out-of-Order Pipeline – Cycle 2a



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | | | | | | | | | | |
| add r2 + r3 → r4 | F | Di | | | | | | | | | | | |
| xor r4 ^ r5 → r6 | | F | | | | | | | | | | | |
| ld [r7] → r4 | | F | | | | | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p10 |
| r5 | p4 |
| r6 | p3 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | no |
| p10 | no |
| p11 | --- |
| p12 | --- |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | p5 | no |
| xor | | no |
| ld | | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ld | p8 | yes | --- | yes | p9 | 0 |
| add | p9 | no | p6 | yes | p10 | 1 |
| | | | | | | |
| | | | | | | |

# Out-of-Order Pipeline – Cycle 2b

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | | | | | | | | | | |
| add r2 + r3 → r4 | F | Di | | | | | | | | | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | | | | | | | |
| ld [r7] → r4 | | F | | | | | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p10 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | no |
| p10 | no |
| p11 | no |
| p12 | --- |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | p5 | no |
| xor | p3 | no |
| ld | | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| add | p9 | no | p6 | yes | p10 | 1 |
| xor | p10 | no | p4 | yes | p11 | 2 |
| | | | | | | |

# Out-of-Order Pipeline – Cycle 2c

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | | | | | | | | | | |
| add r2 + r3 → r4 | F | Di | | | | | | | | | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | | | | | | | |
| ld [r7] → r4 | | F | Di | | | | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | no |
| p10 | no |
| p11 | no |
| p12 | no |

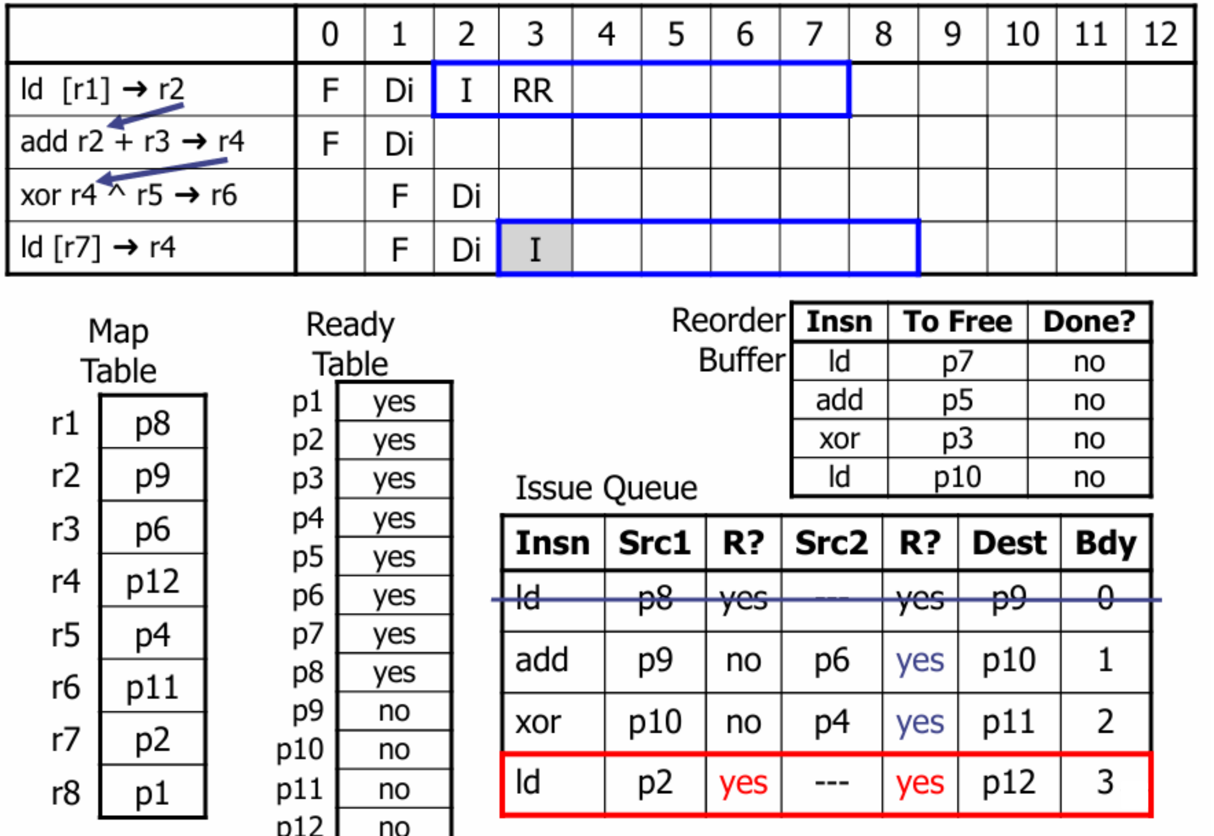
Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | p5 | no |
| xor | p3 | no |
| ld | p10 | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| add | p9 | no | p6 | yes | p10 | 1 |
| xor | p10 | no | p4 | yes | p11 | 2 |
| ld | p2 | yes | --- | yes | p12 | 3 |

# Out-of-Order Pipeline – Cycle 3



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | | | | | | | | | |
| add r2 + r3 → r4 | F | Di | | | | | | | | | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | | | | | | | |
| ld [r7] → r4 | | F | Di | I | | | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | no |
| p10 | no |
| p11 | no |
| p12 | no |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | p5 | no |
| xor | p3 | no |
| ld | p10 | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| add | p9 | no | p6 | yes | p10 | 1 |
| xor | p10 | no | p4 | yes | p11 | 2 |
| ld | p2 | yes | --- | yes | p12 | 3 |

# Out-of-Order Pipeline – Cycle 4

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | | | | | | | | |
| add r2 + r3 → r4 | F | Di | | | | | | | | | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | | | | | | | |
| ld [r7] → r4 | | F | Di | I | RR | | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | yes |
| p10 | no |
| p11 | no |
| p12 | no |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | p5 | no |
| xor | p3 | no |
| ld | p10 | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| add | p9 | yes | p6 | yes | p10 | 1 |
| xor | p10 | no | p4 | yes | p11 | 2 |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

# Out-of-Order Pipeline – Cycle 5a

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | $M_1$ | | | | | | | |
| add r2 + r3 → r4 | F | Di | | | | I | | | | | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | | | | | | | |
| ld [r7] → r4 | | F | Di | I | RR | X | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | yes |
| p10 | yes |
| p11 | no |
| p12 | no |

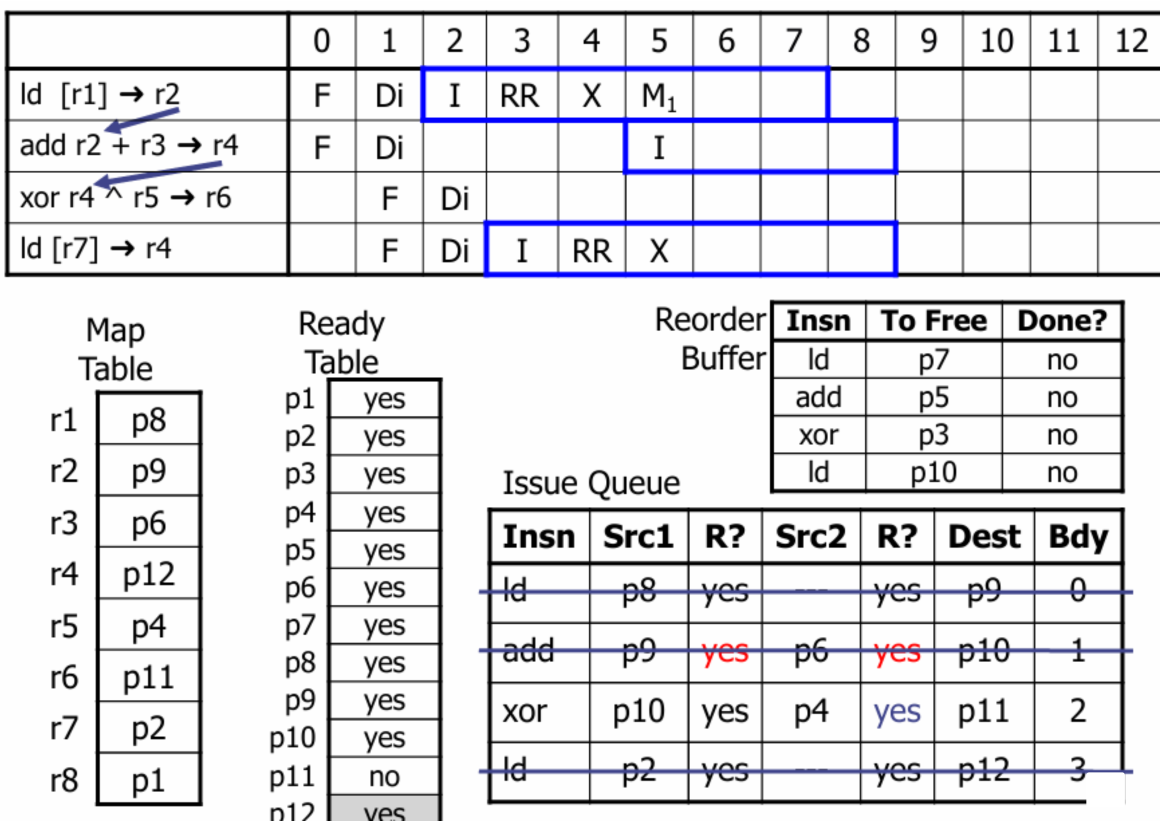
Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | p5 | no |
| xor | p3 | no |
| ld | p10 | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| add | p9 | yes | p6 | yes | p10 | 1 |
| xor | p10 | yes | p4 | yes | p11 | 2 |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

# Out-of-Order Pipeline – Cycle 5b



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | $M_1$ | | | | | | | |
| add r2 + r3 → r4 | F | Di | | | | I | | | | | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | | | | | | | |
| ld [r7] → r4 | | F | Di | I | RR | X | | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | yes |
| p10 | yes |
| p11 | no |
| p12 | yes |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | p5 | no |
| xor | p3 | no |
| ld | p10 | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| ~~add~~ | ~~p9~~ | ~~yes~~ | ~~p6~~ | ~~yes~~ | ~~p10~~ | ~~1~~ |
| xor | p10 | yes | p4 | yes | p11 | 2 |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

# Out-of-Order Pipeline – Cycle 6

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | $M_1$ | $M_2$ | | | | | | |
| add r2 + r3 → r4 | F | Di | | | | I | RR | | | | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | I | | | | | | |
| ld [r7] → r4 | | F | Di | I | RR | X | $M_1$ | | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | yes |
| p10 | yes |
| p11 | yes |
| p12 | yes |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | no |
| add | p5 | no |
| xor | p3 | no |
| ld | p10 | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| ~~add~~ | ~~p9~~ | ~~yes~~ | ~~p6~~ | ~~yes~~ | ~~p10~~ | ~~1~~ |
| xor | p10 | yes | p4 | yes | p11 | 2 |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

# Out-of-Order Pipeline – Cycle 7

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | $M_1$ | $M_2$ | W | | | | | |
| add r2 + r3 → r4 | F | Di | | | | I | RR | X | | | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | I | RR | | | | | |
| ld [r7] → r4 | | F | Di | I | RR | X | $M_1$ | $M_2$ | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | yes |
| p8 | yes |
| p9 | yes |
| p10 | yes |
| p11 | yes |
| p12 | yes |

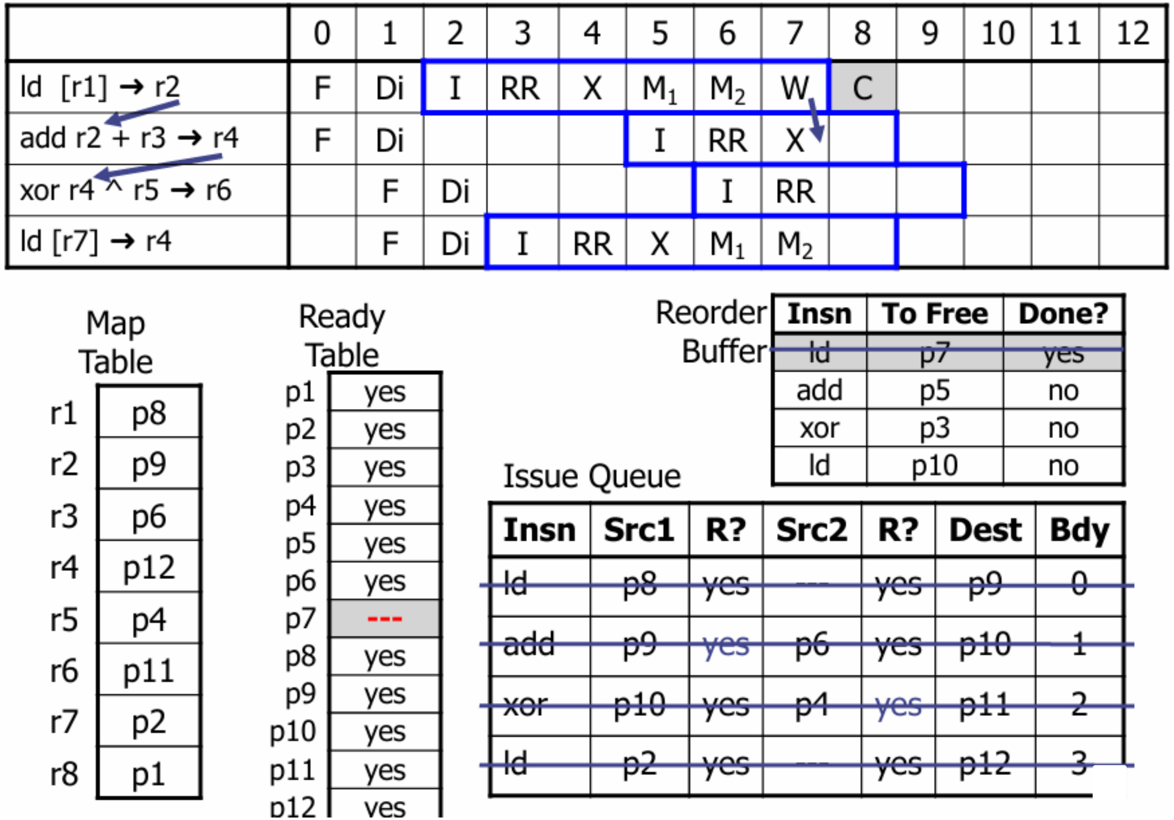
Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ld | p7 | yes |
| add | p5 | no |
| xor | p3 | no |
| ld | p10 | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| ~~add~~ | ~~p9~~ | ~~yes~~ | ~~p6~~ | ~~yes~~ | ~~p10~~ | ~~1~~ |
| ~~xor~~ | ~~p10~~ | ~~yes~~ | ~~p4~~ | ~~yes~~ | ~~p11~~ | ~~2~~ |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

# Out-of-Order Pipeline – Cycle 8a



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | $M_1$ | $M_2$ | W | C | | | | |
| add r2 + r3 → r4 | F | Di | | | | I | RR | X | | | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | I | RR | | | | | |
| ld [r7] → r4 | | F | Di | I | RR | X | $M_1$ | $M_2$ | | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | --- |
| p8 | yes |
| p9 | yes |
| p10 | yes |
| p11 | yes |
| p12 | yes |

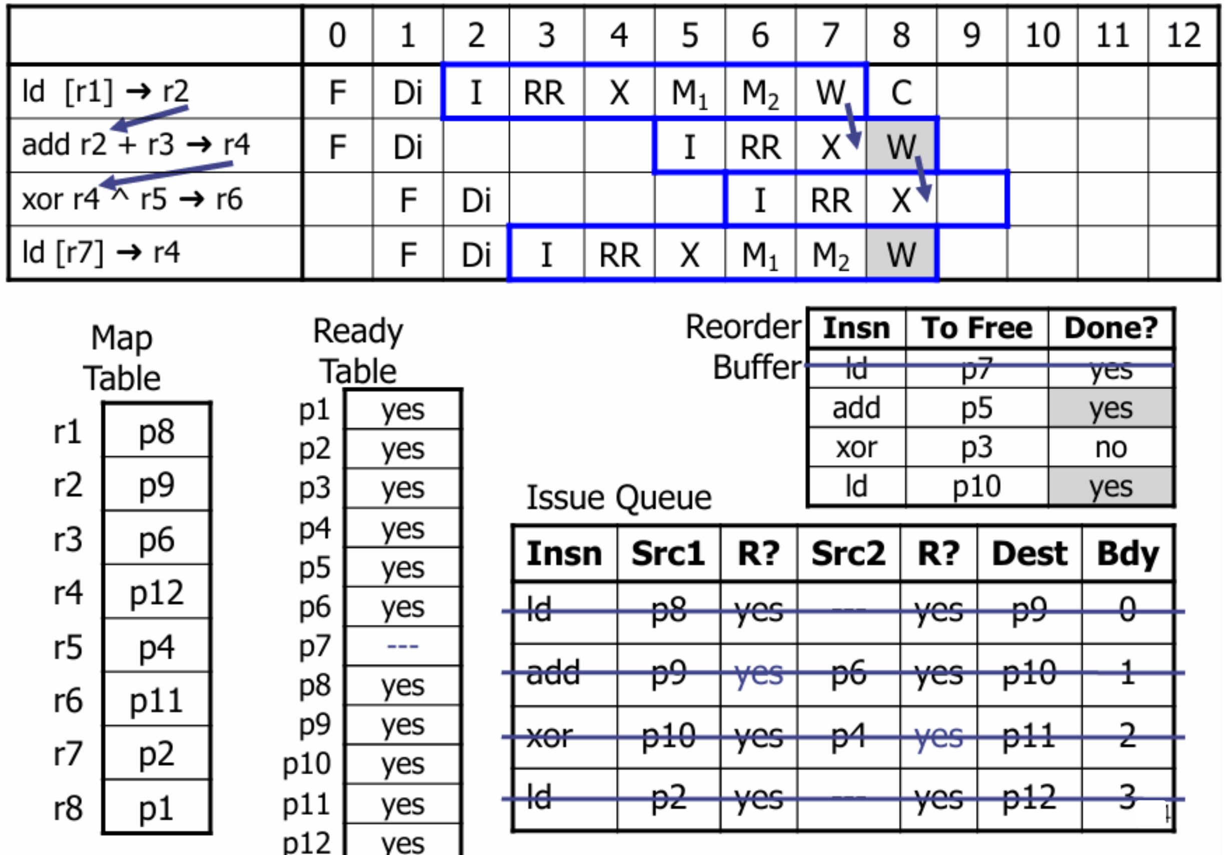
Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ~~ld~~ | ~~p7~~ | ~~yes~~ |
| add | p5 | no |
| xor | p3 | no |
| ld | p10 | no |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| ~~add~~ | ~~p9~~ | ~~yes~~ | ~~p6~~ | ~~yes~~ | ~~p10~~ | ~~1~~ |
| ~~xor~~ | ~~p10~~ | ~~yes~~ | ~~p4~~ | ~~yes~~ | ~~p11~~ | ~~2~~ |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

# Out-of-Order Pipeline – Cycle 8b



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | $M_1$ | $M_2$ | W | C | | | | |
| add r2 + r3 → r4 | F | Di | | | | I | RR | X | W | | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | I | RR | X | | | | |
| ld [r7] → r4 | | F | Di | I | RR | X | $M_1$ | $M_2$ | W | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | yes |
| p6 | yes |
| p7 | --- |
| p8 | yes |
| p9 | yes |
| p10 | yes |
| p11 | yes |
| p12 | yes |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ~~ld~~ | ~~p7~~ | ~~yes~~ |
| add | p5 | yes |
| xor | p3 | no |
| ld | p10 | yes |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| ~~add~~ | ~~p9~~ | ~~yes~~ | ~~p6~~ | ~~yes~~ | ~~p10~~ | ~~1~~ |
| ~~xor~~ | ~~p10~~ | ~~yes~~ | ~~p4~~ | ~~yes~~ | ~~p11~~ | ~~2~~ |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

## Out-of-Order Pipeline – Cycle 9a

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | $M_1$ | $M_2$ | W | C | | | | |
| add r2 + r3 → r4 | F | Di | | | | I | RR | X | W | C | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | I | RR | X | | | | |
| ld [r7] → r4 | | F | Di | I | RR | X | $M_1$ | $M_2$ | W | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | --- |
| p6 | yes |
| p7 | --- |
| p8 | yes |
| p9 | yes |
| p10 | yes |
| p11 | yes |
| p12 | yes |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ~~ld~~ | ~~p7~~ | ~~yes~~ |
| ~~add~~ | ~~p5~~ | ~~yes~~ |
| xor | p3 | no |
| ld | p10 | yes |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| ~~add~~ | ~~p9~~ | ~~yes~~ | ~~p6~~ | ~~yes~~ | ~~p10~~ | ~~1~~ |
| ~~xor~~ | ~~p10~~ | ~~yes~~ | ~~p4~~ | ~~yes~~ | ~~p11~~ | ~~2~~ |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

# Out-of-Order Pipeline – Cycle 9b

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | $M_1$ | $M_2$ | W | C | | | | |
| add r2 + r3 → r4 | F | Di | | | | I | RR | X | W | C | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | I | RR | X | W | | | |
| ld [r7] → r4 | | F | Di | I | RR | X | $M_1$ | $M_2$ | W | | | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | yes |
| p4 | yes |
| p5 | --- |
| p6 | yes |
| p7 | --- |
| p8 | yes |
| p9 | yes |
| p10 | yes |
| p11 | yes |
| p12 | yes |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ~~ld~~ | ~~p7~~ | ~~yes~~ |
| ~~add~~ | ~~p5~~ | ~~yes~~ |
| xor | p3 | yes |
| ld | p10 | yes |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| ~~add~~ | ~~p9~~ | ~~yes~~ | ~~p6~~ | ~~yes~~ | ~~p10~~ | ~~1~~ |
| ~~xor~~ | ~~p10~~ | ~~yes~~ | ~~p4~~ | ~~yes~~ | ~~p11~~ | ~~2~~ |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

## Out-of-Order Pipeline – Cycle 10

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | $M_1$ | $M_2$ | W | C | | | | |
| add r2 + r3 → r4 | F | Di | | | | I | RR | X | W | C | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | I | RR | X | W | C | | |
| ld [r7] → r4 | | F | Di | I | RR | X | $M_1$ | $M_2$ | W | | C | | |

Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | --- |
| p4 | yes |
| p5 | --- |
| p6 | yes |
| p7 | --- |
| p8 | yes |
| p9 | yes |
| p10 | --- |
| p11 | yes |
| p12 | yes |

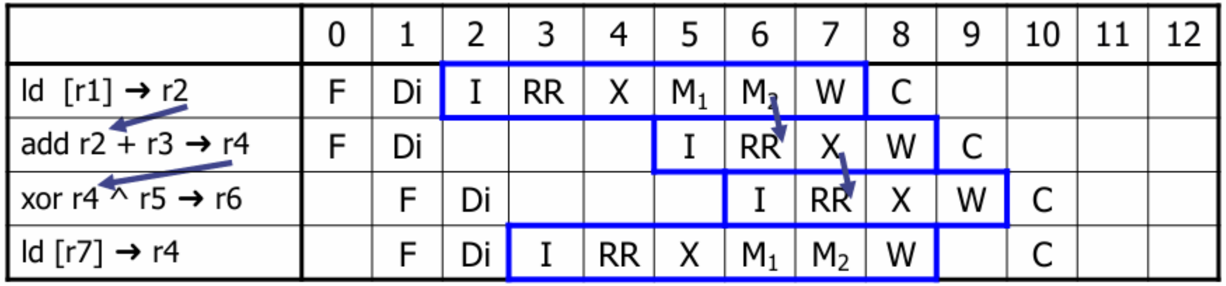
Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ~~ld~~ | ~~p7~~ | ~~yes~~ |
| ~~add~~ | ~~p5~~ | ~~yes~~ |
| ~~xor~~ | ~~p3~~ | ~~yes~~ |
| ~~ld~~ | ~~p10~~ | ~~yes~~ |

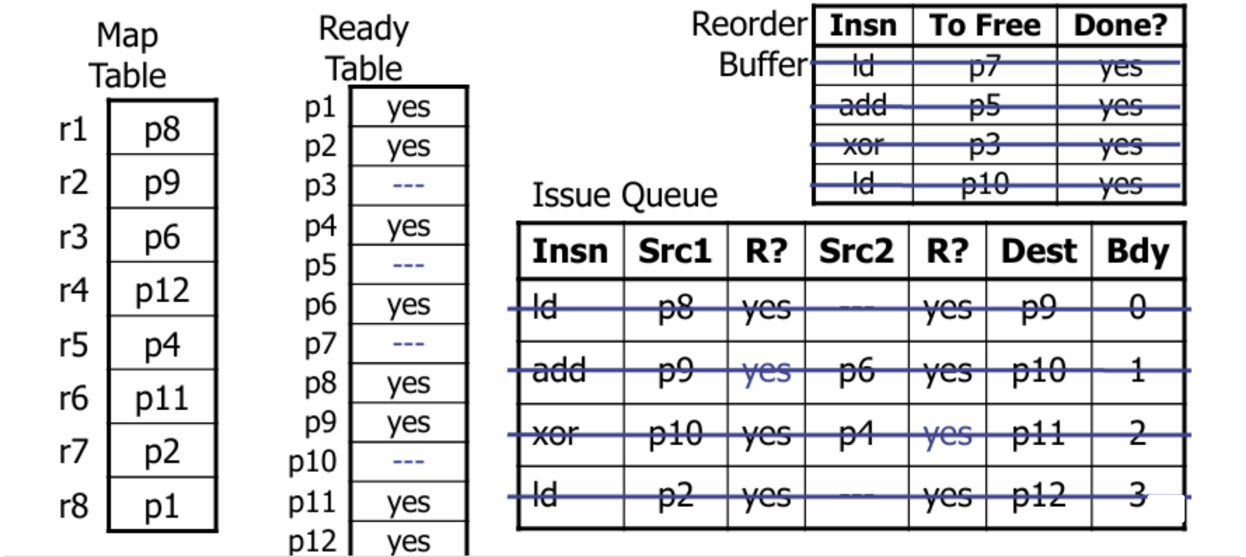
Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| ~~add~~ | ~~p9~~ | ~~yes~~ | ~~p6~~ | ~~yes~~ | ~~p10~~ | ~~1~~ |
| ~~xor~~ | ~~p10~~ | ~~yes~~ | ~~p4~~ | ~~yes~~ | ~~p11~~ | ~~2~~ |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

# Out-of-Order Pipeline – Done!



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ld [r1] → r2 | F | Di | I | RR | X | $M_1$ | $M_2$ | W | C | | | | |
| add r2 + r3 → r4 | F | Di | | | | I | RR | X | W | C | | | |
| xor r4 ^ r5 → r6 | | F | Di | | | | I | RR | X | W | C | | |
| ld [r7] → r4 | | F | Di | I | RR | X | $M_1$ | $M_2$ | W | | C | | |



Map Table

| | |
|---|---|
| r1 | p8 |
| r2 | p9 |
| r3 | p6 |
| r4 | p12 |
| r5 | p4 |
| r6 | p11 |
| r7 | p2 |
| r8 | p1 |

Ready Table

| | |
|---|---|
| p1 | yes |
| p2 | yes |
| p3 | --- |
| p4 | yes |
| p5 | --- |
| p6 | yes |
| p7 | --- |
| p8 | yes |
| p9 | yes |
| p10 | --- |
| p11 | yes |
| p12 | yes |

Reorder Buffer

| Insn | To Free | Done? |
|---|---|---|
| ~~ld~~ | ~~p7~~ | ~~yes~~ |
| ~~add~~ | ~~p5~~ | ~~yes~~ |
| ~~xor~~ | ~~p3~~ | ~~yes~~ |
| ~~ld~~ | ~~p10~~ | ~~yes~~ |

Issue Queue

| Insn | Src1 | R? | Src2 | R? | Dest | Bdy |
|---|---|---|---|---|---|---|
| ~~ld~~ | ~~p8~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p9~~ | ~~0~~ |
| ~~add~~ | ~~p9~~ | ~~yes~~ | ~~p6~~ | ~~yes~~ | ~~p10~~ | ~~1~~ |
| ~~xor~~ | ~~p10~~ | ~~yes~~ | ~~p4~~ | ~~yes~~ | ~~p11~~ | ~~2~~ |
| ~~ld~~ | ~~p2~~ | ~~yes~~ | ~~---~~ | ~~yes~~ | ~~p12~~ | ~~3~~ |

## Conclusion

- OoO Everywhere
- Apple Cyclone core
  - part of A7 chip, launched in iPhone 5S in 2013
  - issue 6 insns per cycle
  - 192-entry ROB
  - 4 integer ALUs
  - 2 Load/Store units
  - 14-19 cycle branch misprediction penalty
  - https://www.anandtech.com/show/7910/apples-cyclone-microarchitecture-detailed