# EEL 6764 Principles of Computer Architecture
# Instruction-Level Parallelism

Dr Hao Zheng
Dept. of Comp Sci & Eng
U of South Florida

# Instruction Level Parallelism

# Instruction-Level Parallelism

- → Overlap executions of multiple instructions
- → When exploiting instruction-level parallelism, goal is to maximize CPI
  - → Pipeline CPI =

    Ideal pipeline CPI  
    + Structural stalls  
    + Data hazard stalls  
    + Control stalls
- → Two approaches to exploiting ILP
  - → Static scheduling
  - → Dynamic scheduling

# Dependences

- Parallelism with **basic block** is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches
- Increase ILP – Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)
- Challenges: Data dependency
  - Instruction *j* is data dependent on instruction *i* if
    - Instruction *i* produces a result that may be used by instruction *j*
    - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*
- Dependent instructions cannot be executed simultaneously

# Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall
- **Data dependence** conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect
  - SD R1, 45(R5)
  - LD R2, 90(R7)

# Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, *but is a problem when reordering instructions*
  - **Antidependence**: instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - **Output dependence**: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved

- To resolve, use **renaming** techniques

# Data Hazards

- A **hazard** exists between two dependent instructions that are close in a program such that overlapping their executions would change the order of accesses to the operand involved in the dependence.

- **Data Hazards**
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)

# Control Dependences

- **Control Dependence** determines ordering of instruction i with respect to a branch instruction
  - Instruction control-dependent on a branch cannot be moved before the branch so that its execution is no longer controller by the branch
  - An instruction not control-dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

```
if p1 { s1; }  // s1 control-dependent on p1
if p2 { s2; }  // s2 control-dependent on p2
```

# Control Dependence – Examples

Example 1:

```
    ADD x2, x3, x4
    BEQ x2, x0, L
    LD  x1, 0(x2)
L:  ...
```

Example 2:

```
    ADD x1, x2, x3
    BEQ x4, x0, L
    SUB x1, x5, x6
L:  ...
    OR  x7, x1, x8
```

- LD cannot be move above branch
- OR instruction dependent on ADD and SUB
  - Cannot move SUB before the branch
- Reorder instruction if it does not change program semantics

# Control Dependence – Examples (1)

Example 3

```
    ADD  x1, x2, x3
    BEQ  x12, x0, skip
    SUB  x4, x5, x6
    ADD  x5, x4, x9
skip:   ...
    OR   x7, x8, x9
```

- → Reorder instruction if it does not change program semantics
- → Example 3: ok to move sub above BEQ
  - → if x4 is not used after label "skip"

# 3.2 Compiler Techniques for Exposing ILP

# Pipeline Scheduling

- Separate dependent instruction from the source instruction by the **pipeline latency** of the source instruction
  - Pipeline latency – cycles to separate two dependent instructions to avoid hazards
- By inserting NOP (bubbles) – pipeline stalls
- By inserting independent instructions - **Scheduling**
  - Statically
  - Dynamically
- Scheduling relies on
  - ILP in programs
  - latencies of functional units

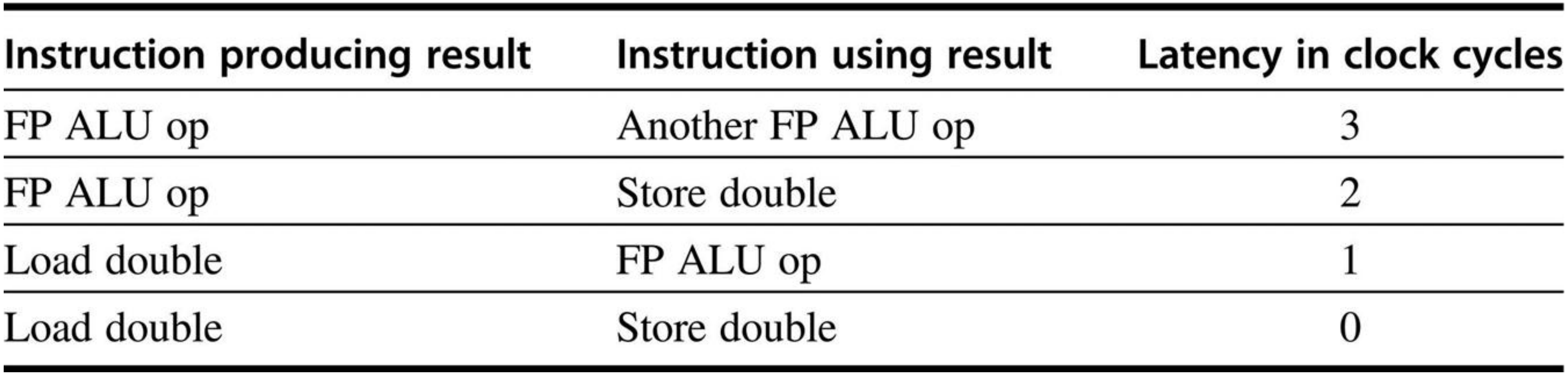
# Static Scheduling

- → Transform and re-arrange the code to reduce pipeline stalls

- → Example:

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

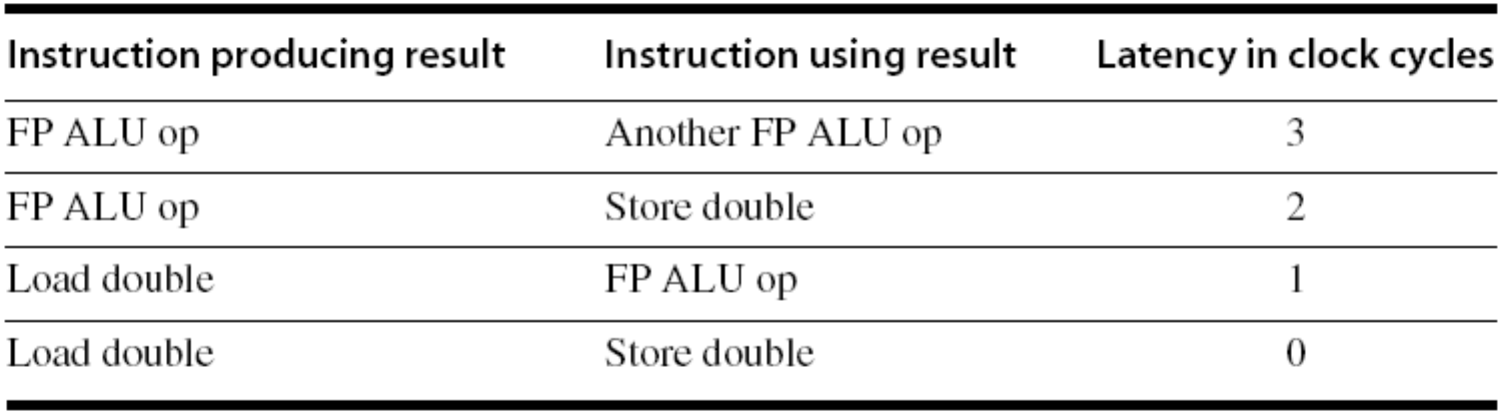
Assume branch delay = 1 cycle



| 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 |

# Static Scheduling (1)

```
Loop:   fld     f0, 0(x1)
        stall
        fdd.d   f4, f0, f2
        stall
        stall
        fsd     f4, 0(x1)
        addi    x1, x1, -8
        bne     x1, x2, Loop
```

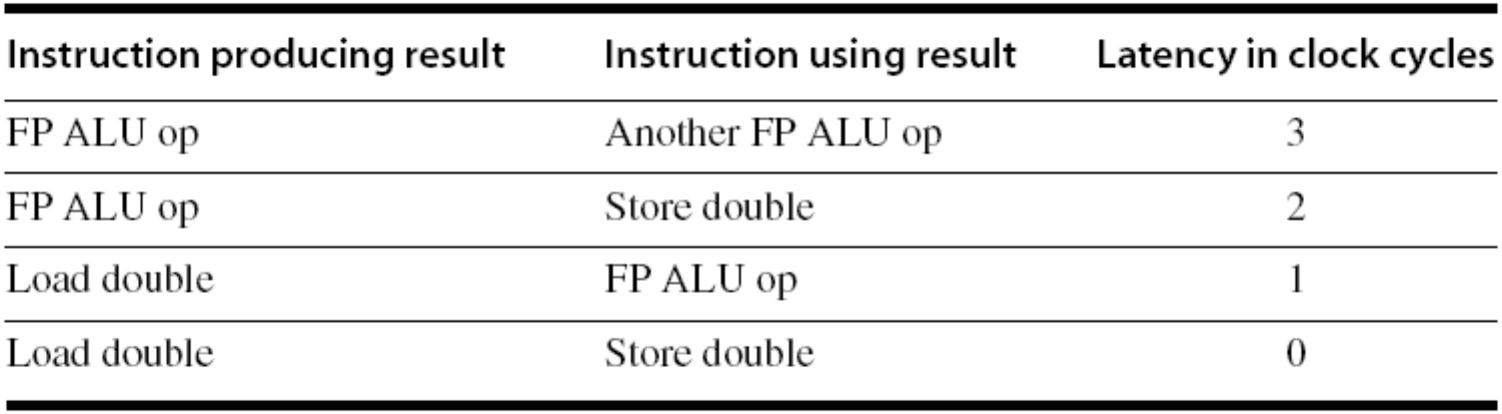


| Instruction producing result | Instruction using result | Latency in clock cycles |
|---|---|---|
| FP ALU op | Another FP ALU op | 3 |
| FP ALU op | Store double | 2 |
| Load double | FP ALU op | 1 |
| Load double | Store double | 0 |

# Static Scheduling (2)

```
Loop:   fld     f0, 0(x1)
        addi    x1, x1, -8
        fdd.d   f4, f0, f2
        stall
        stall
        fsd     f4, 8(x1)
        bne     x1, x2, Loop
```

- → ILP in basic blocks is limited
- → Loop unrolling to increase ILP



| 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 |

# Loop Unrolling

- Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

```
Loop:   fld     f0, 0(x1)
        fadd.d  f4, f0, f2
        fsd     f4, 0(R1)  ;drop DADDUI & BNE
        fld     f6, -8(x1)
        fadd.d  f8, F6, f2
        fsd     f8, -8(x1)  ;drop addi & BNE
        fld     f0, -16(x1)
        fadd.d  f12, f0, f2
        fsd     f12,- 16(x1)  ;drop addi & BNE
        fld     f14, -24(x1)
        fadd.d  f16, f14, f2
        fsd     f16, -24(x1)
        addi    x1, x1, -32
        bne     x1, x2, Loop
```

Run in 26 cycles without scheduling

Number of live registers increase vs. original loop

- Avoid dependence
- Increase ILP

# Loop Unrolling/Pipeline Scheduling

→ Pipeline schedule the unrolled loop

```
Loop:   fld     f0, 0(x1)
        fld     f6, −8(x1)
        fld     f0, −16(x1)
        fld     f14, −24(x1)
        fadd.d  f4, f0, f2
        fadd.d  f8, f6, f2
        fadd.d  f12, f0, f2
        fadd.d  f16, f14, f2
        fsd     f4, 0(x1)
        fsd     f8, −8(x1)
        fsd     f12, 16(x1)
        fsd     f16, 8(x1)
        addi    x1, x1, −32
        bne     x1, x2, Loop
```

⇒ Run in 14 cycles after scheduling

⇒ 3.5 cycles per iteration

# Dealing with Unknown Loop Bound

- Number of iterations = *n*, *// n is unknown*
- Goal: make *k* copies of the loop body
- Solution: generate pair of loops:
  - First executes *n* mod *k* times of original loop body
  - Second executes *n* / *k* times of unrolled loop body by *k* times.
  - "Strip mining"

## Limitations of Loop unrolling

- Cross loop iteration dependence
- Code size increase
- Register shortfall

# 3.4 Dynamic Scheduling

# Dynamic Scheduling (Section 3.4, 3.5)

- In-Order pipeline
  - Issue, execution, and completion of instructions follow program order
- Pipeline stalls if there is data dependence
- But, independent instructions are stalled too.

```
DIV.D     F0, F2, F4
ADD.D     F10 F0, F8
SUB.D     F12, F8, F14
```

- Static scheduling may not identify some dependences
  - e.g. dependencies through memory addresses

# Dynamic Scheduling (Section 3.4, 3.5)

- Hardware rearrange order of instructions
  - To reduce stalls while maintaining data flow and exception

- Advantages:
  - Compiler doesn’t need to have knowledge of microarchitecture
  - Handles cases where dependencies are unknown at compile time
  - Tolerate unpredictable delays in memory accesses, FP operations, etc.

- Disadvantage:
  - Substantial increase in hardware complexity
  - Complicates exceptions

# Dynamic Scheduling

- ID stage is divided into
  - Issue – decode instruction, and check for S-hazards
  - Read Operands – wait until no D-hazards, then read operands
- Instructions pass Issue in-order
- They are stalled or bypass each other in Read Operands
  - Enter Execute when operands are ready
  - Enter Execute out-of-order

# Dynamic Scheduling

- Dynamic scheduling implies:
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion

- Creates the possibility for WAR and WAW hazards

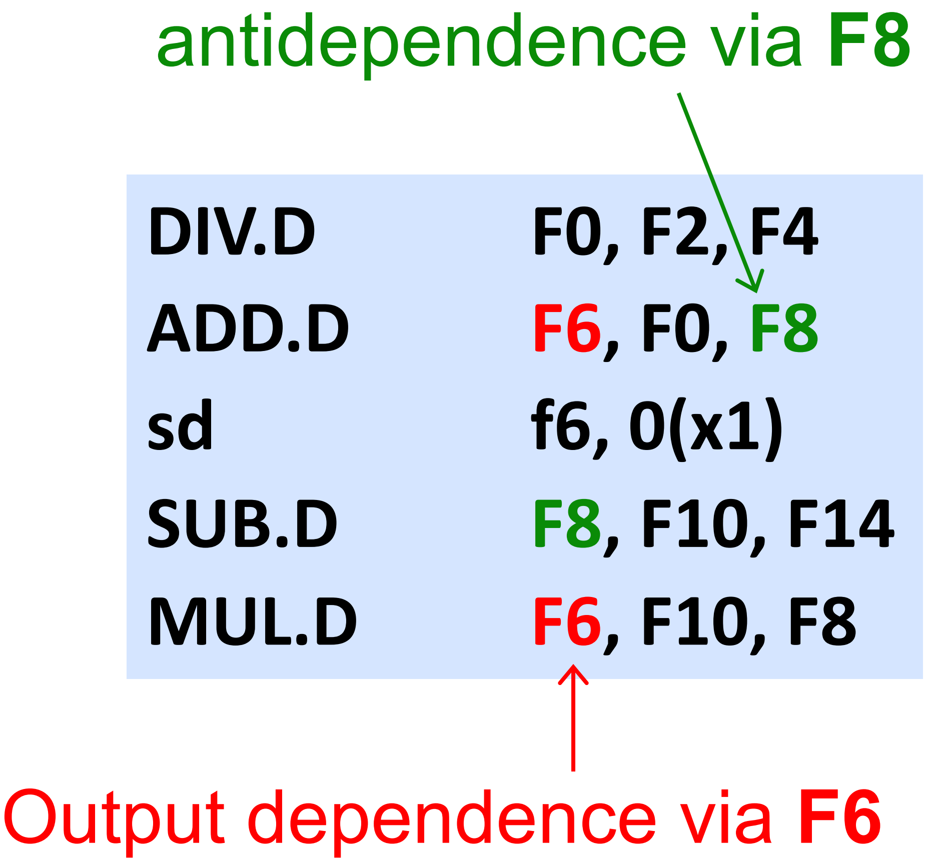
```
DIV.D          F0, F2, F4
ADD.D          F6, F0, F8
SUB.D          F8, F10, F14
MUL.D          F6, F10, F8
```

- WAR/WAW hazards can be avoided by **register renaming**

# Dynamic Scheduling

- Tomasulo's Approach
  - Tracks when operands are available
    - To eliminate RAW hazards
  - Introduces register renaming in hardware
    - To eliminate WAW and WAR hazards

# Register Renaming



antidependence via **F8**

```
DIV.D      F0, F2, F4
ADD.D      F6, F0, F8
sd         f6, 0(x1)
SUB.D      F8, F10, F14
MUL.D      F6, F10, F8
```

Output dependence via **F6**

# Register Renaming

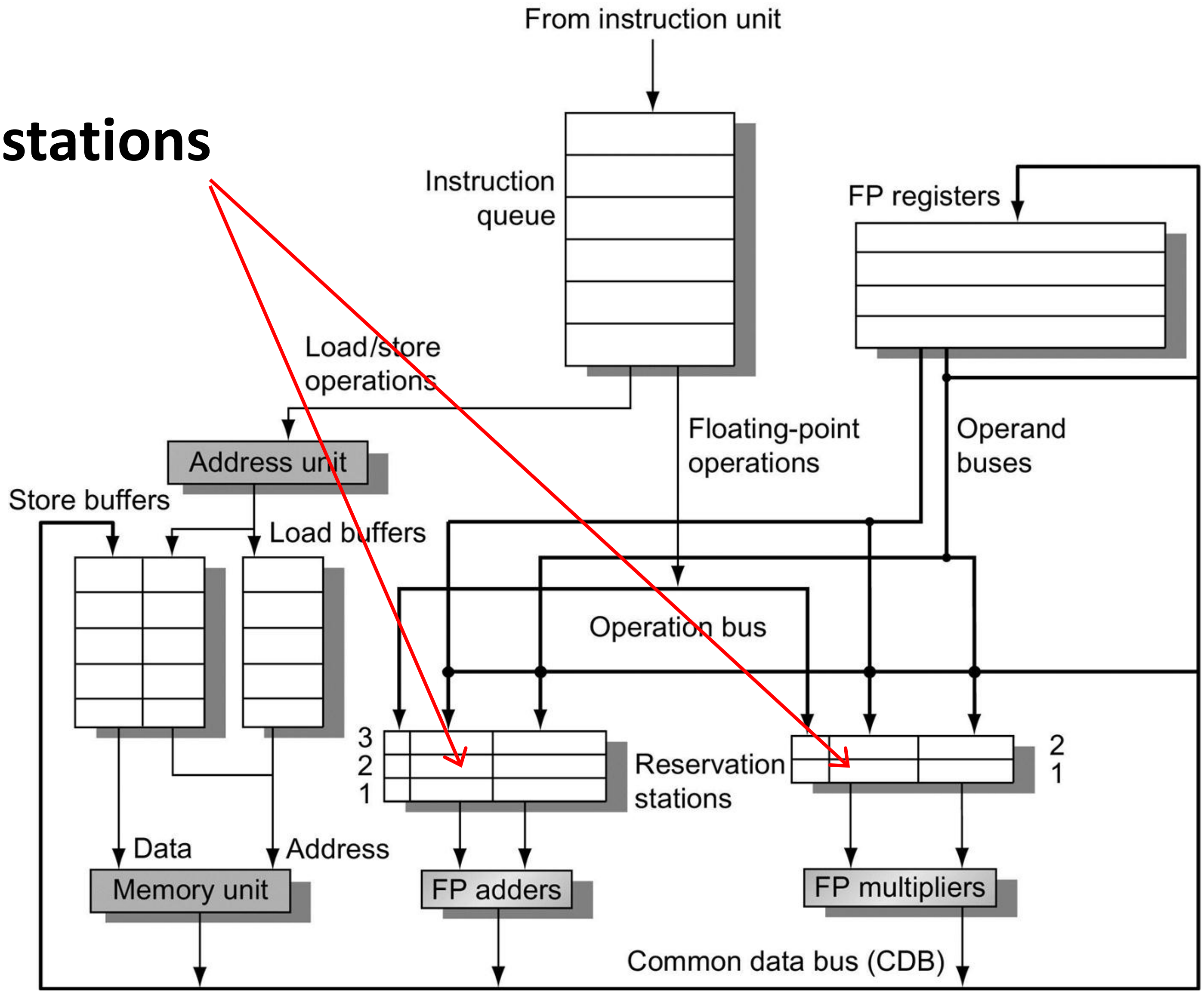
- → Example: assume two additional registers S and T

```
DIV.D      F0, F2, F4
ADD.D      S, F0, F8      ; F6 -> S
sd         S, 0(x1)       ; F6 -> S
SUB.D      T, F10, F14    ; F8 -> T
MUL.D      F6, F10, T     ; F8 -> T
```

- → Now only RAW hazards remain, which must be strictly ordered

# Tomasulo's Algorithm – Structure

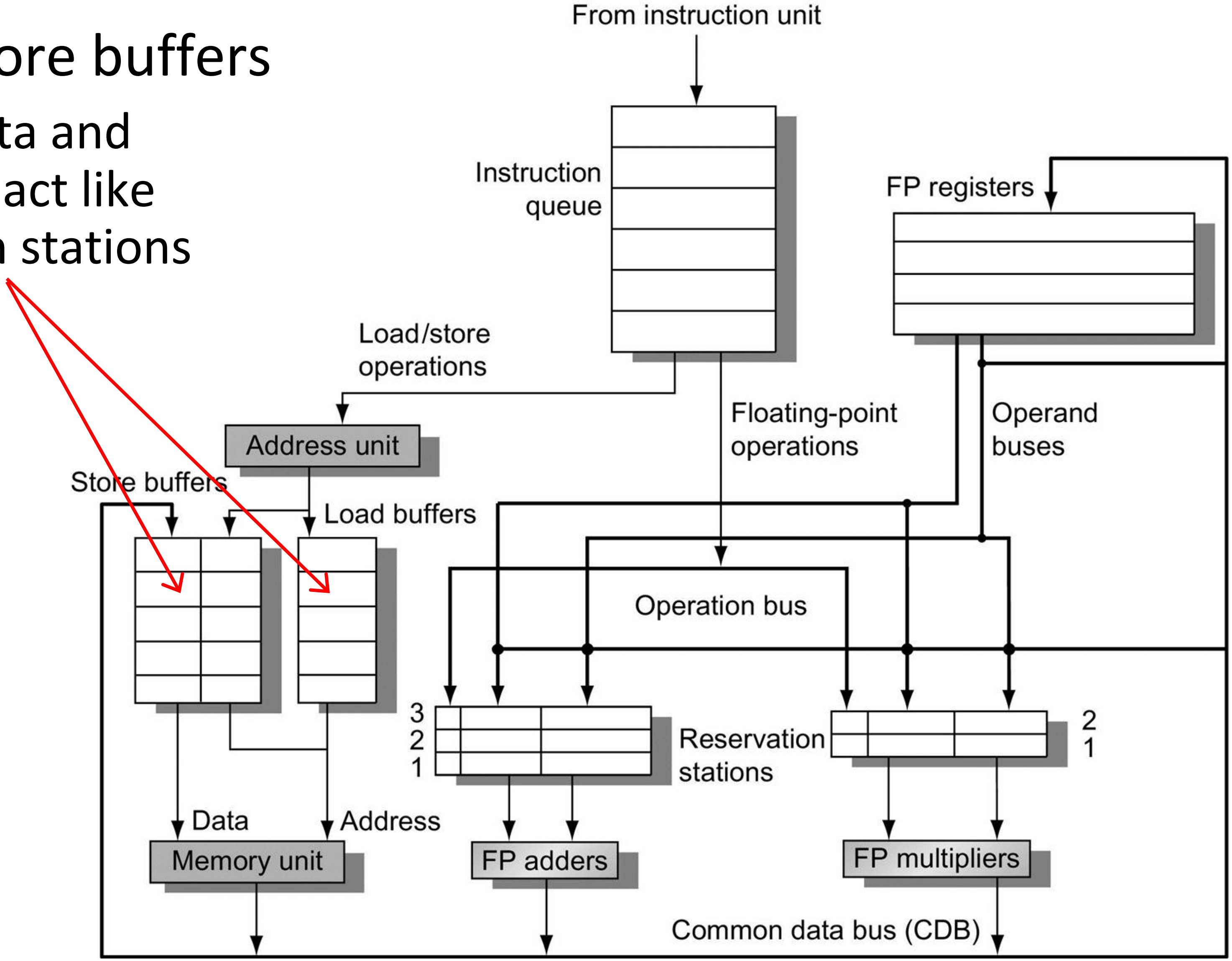
## Reservation stations



# Tomasulo’s Algorithm – Structure

Load and store buffers

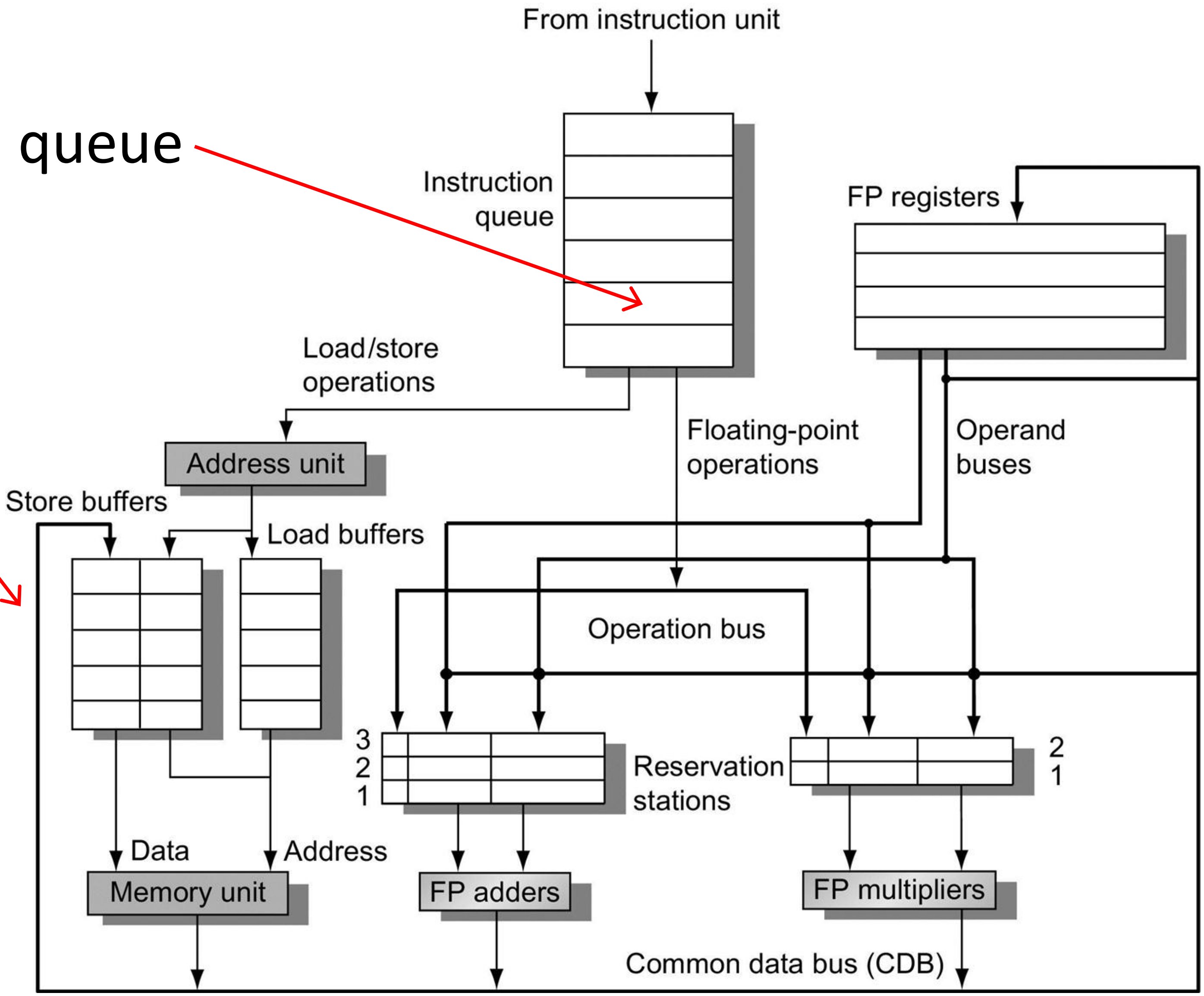
→ Contain data and addresses, act like reservation stations



# Tomasulo’s Algorithm – Structure

→ Instruction queue

→ CDB

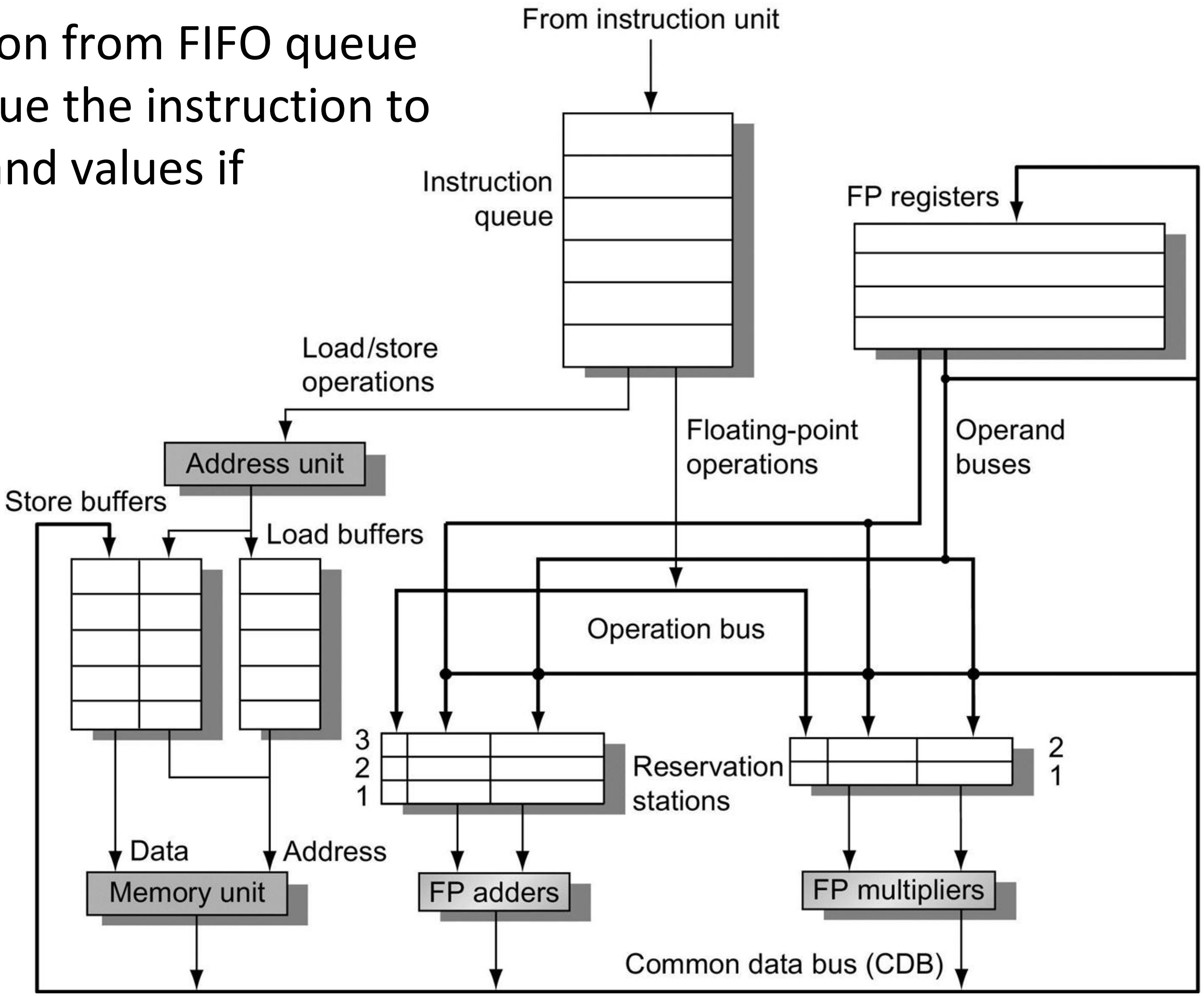


# Register Renaming By Reservation Stations

- Reservation stations (RS) buffer
  - The instruction opcode
  - Buffered operand values (when available)
  - Or, reservation station number of instruction providing the operand values (producer instruction)
- RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
- Pending instructions designate the RS to which they will send their output
- Result values broadcast on a result bus, common data bus (CDB)
- Only the last output updates the register file
- As instructions are issued, the register specifiers are renamed with the reservation station
- May be more reservation stations than registers

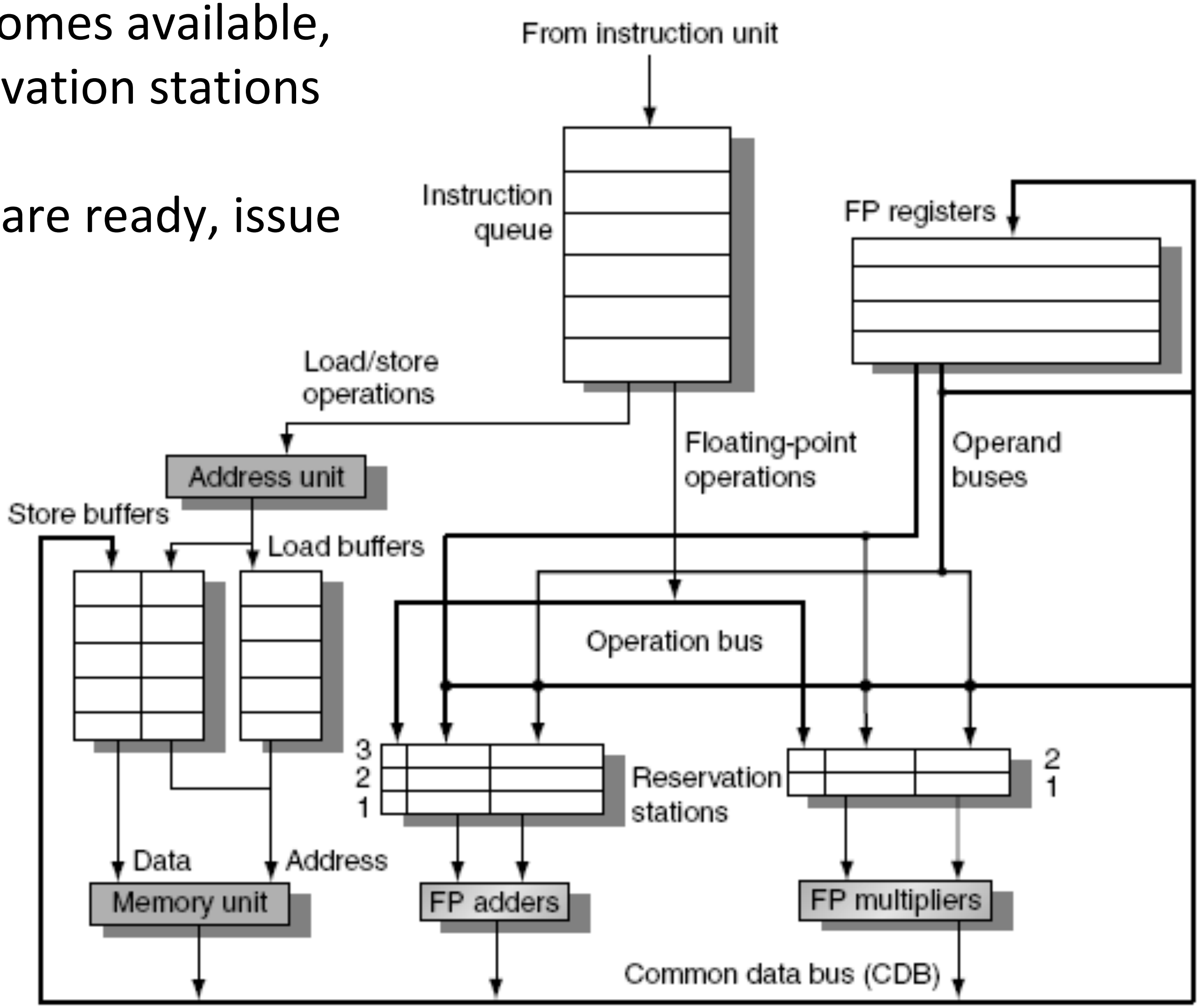
# Tomasulo’s Algorithm – Issue

- Get next instruction from FIFO queue
- If available RS, issue the instruction to the RS with operand values if available
- If operand values not available, stall the instruction
- Renaming in this step



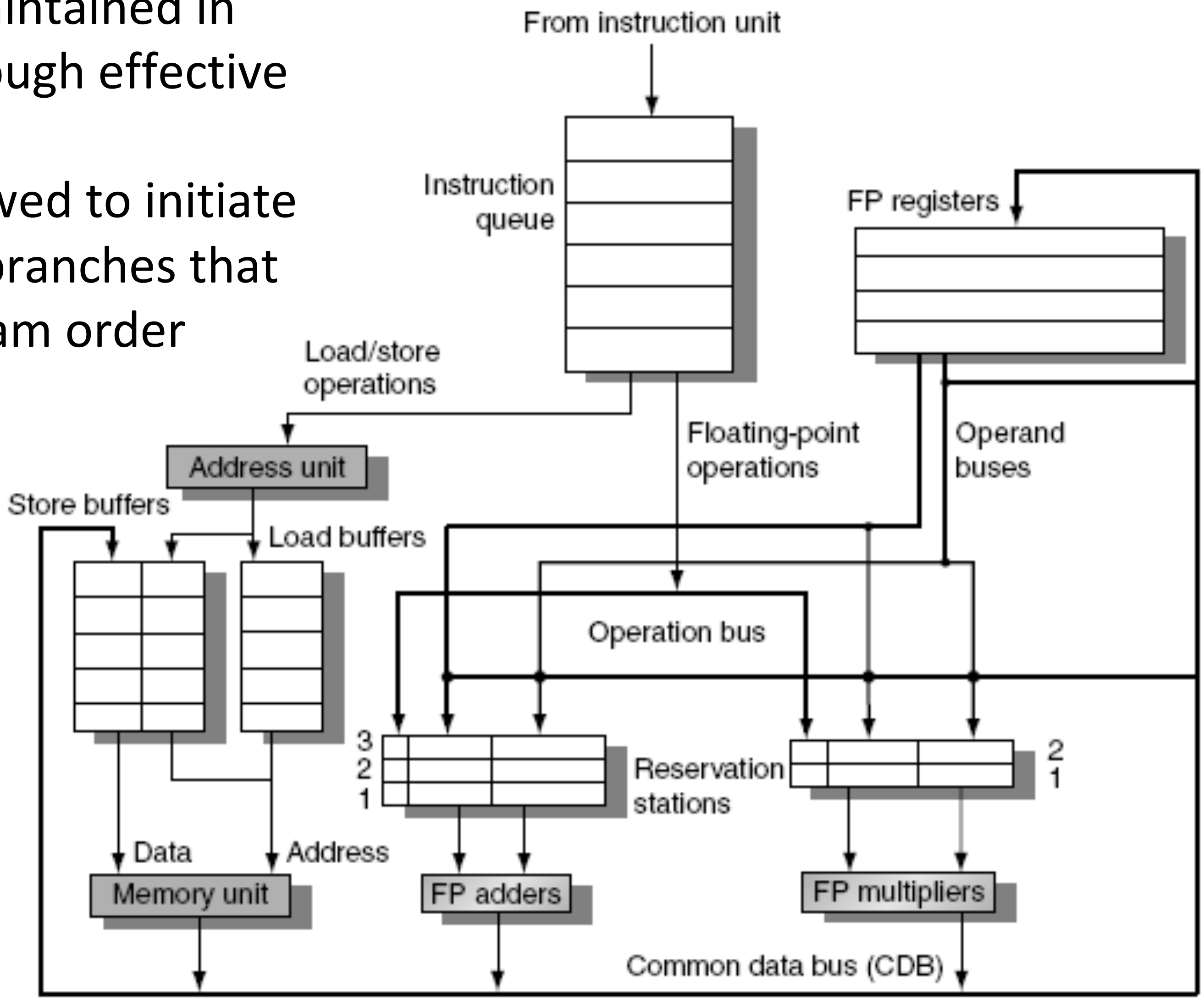
# Tomasulo's Algorithm – Execute

- When operand becomes available, store it in any reservation stations waiting for it
- When all operands are ready, issue the instruction



# Tomasulo's Algorithm – Execute (1)

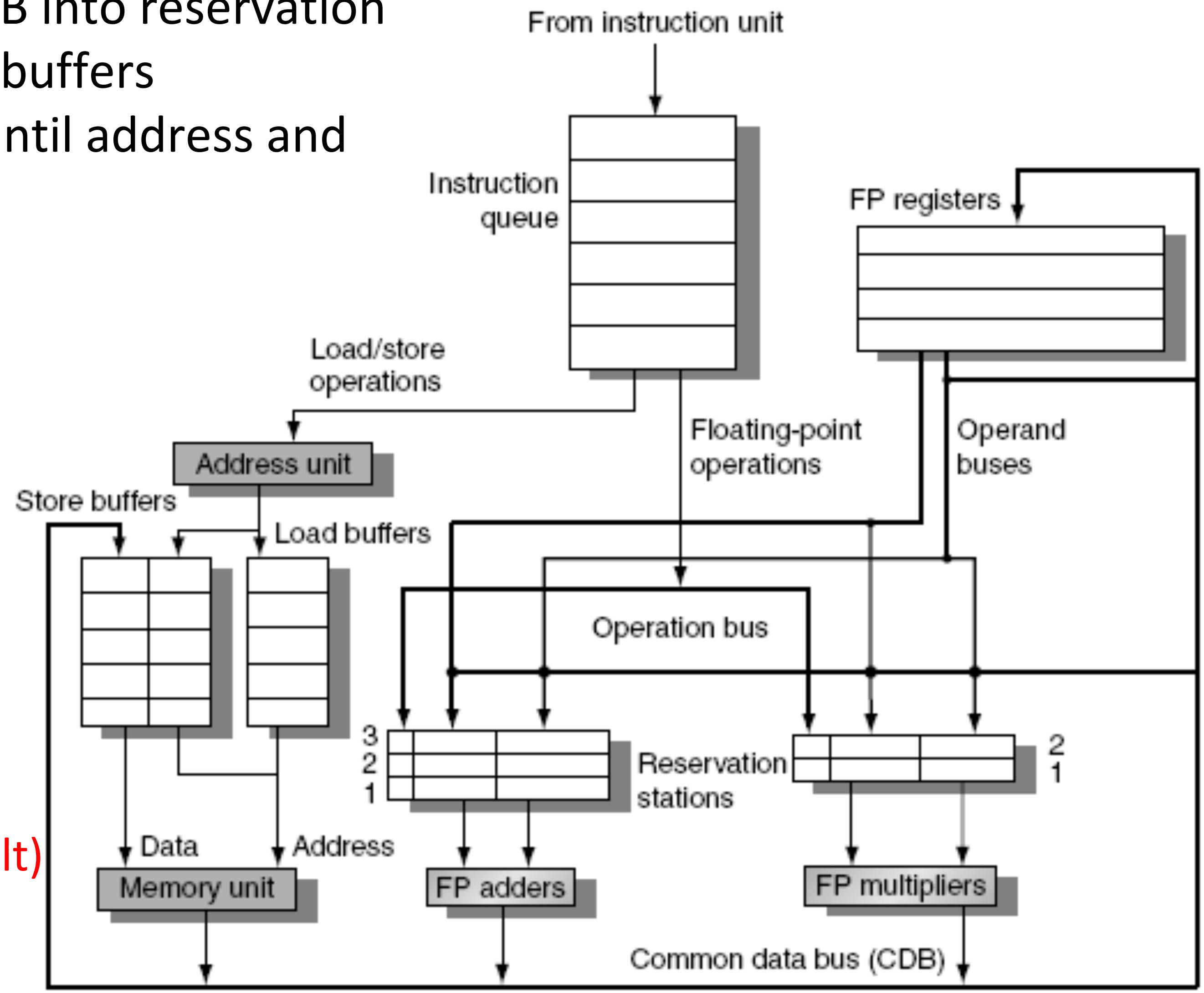
- Loads and store maintained in program order through effective address
- No instruction allowed to initiate execution until all branches that proceed it in program order have completed



# Tomasulo’s Algorithm – Write Result

- Write result on CDB into reservation stations and store buffers
- Stores must wait until address and value are received

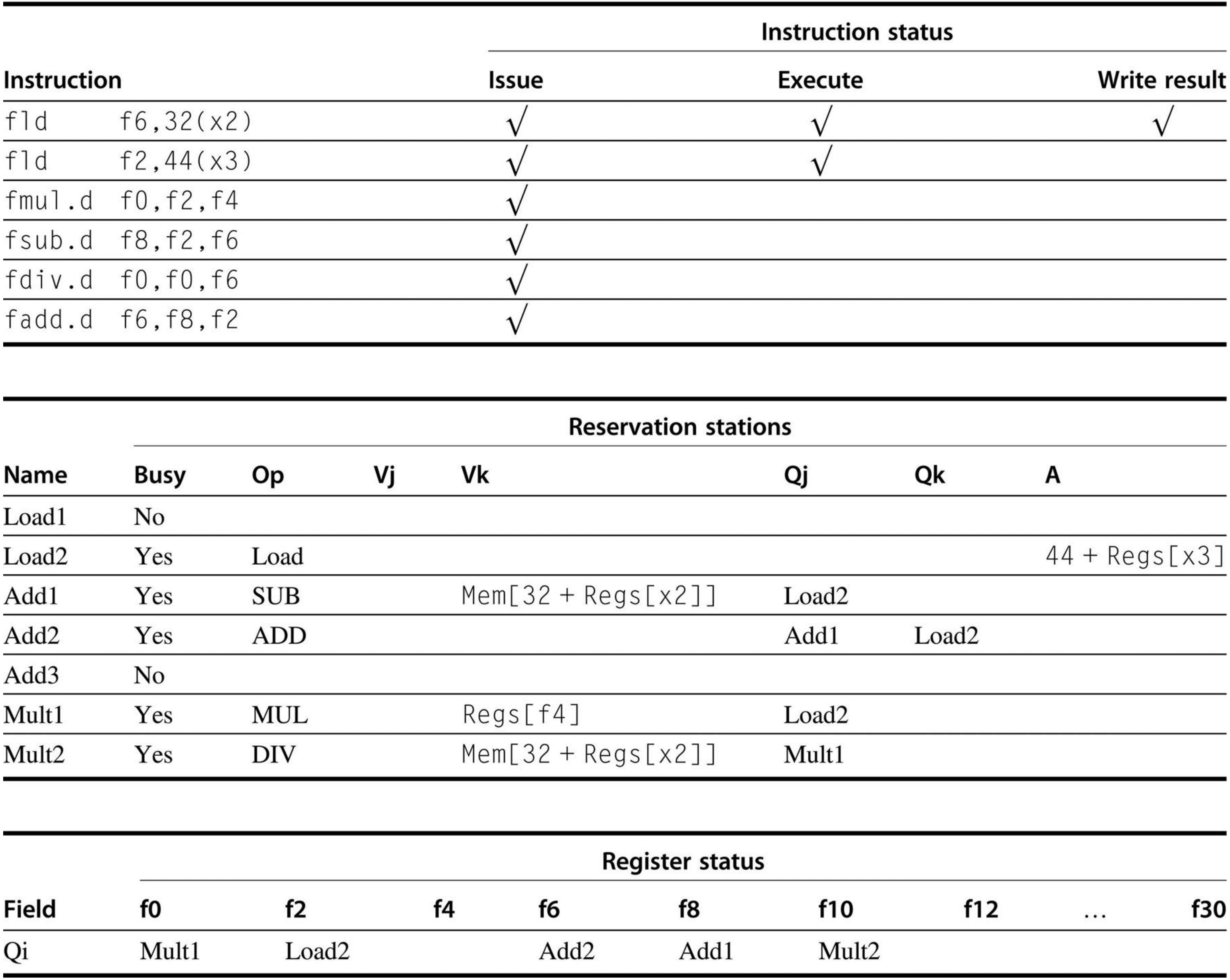
(rs#, result)



# Tomasulo's Algorithm – Some Notes

- Each RS has an unique ID
- Each register is tagged with RS#
  - Will be updated by instruction in RS#
- Each RS, store buffer tagged with RS# for unavailable operands
  - Tags refer to FUs or buffers that produce operands
  - Tags = producer ID
- FUs or memory (producer) put (RS#, results) on CDB
- RS, registers, store buffers (consumer) monitor the CDB,
  - Get the data where its tag matches tag on CDB
- Forwarding through broadcast on CDB

# Example



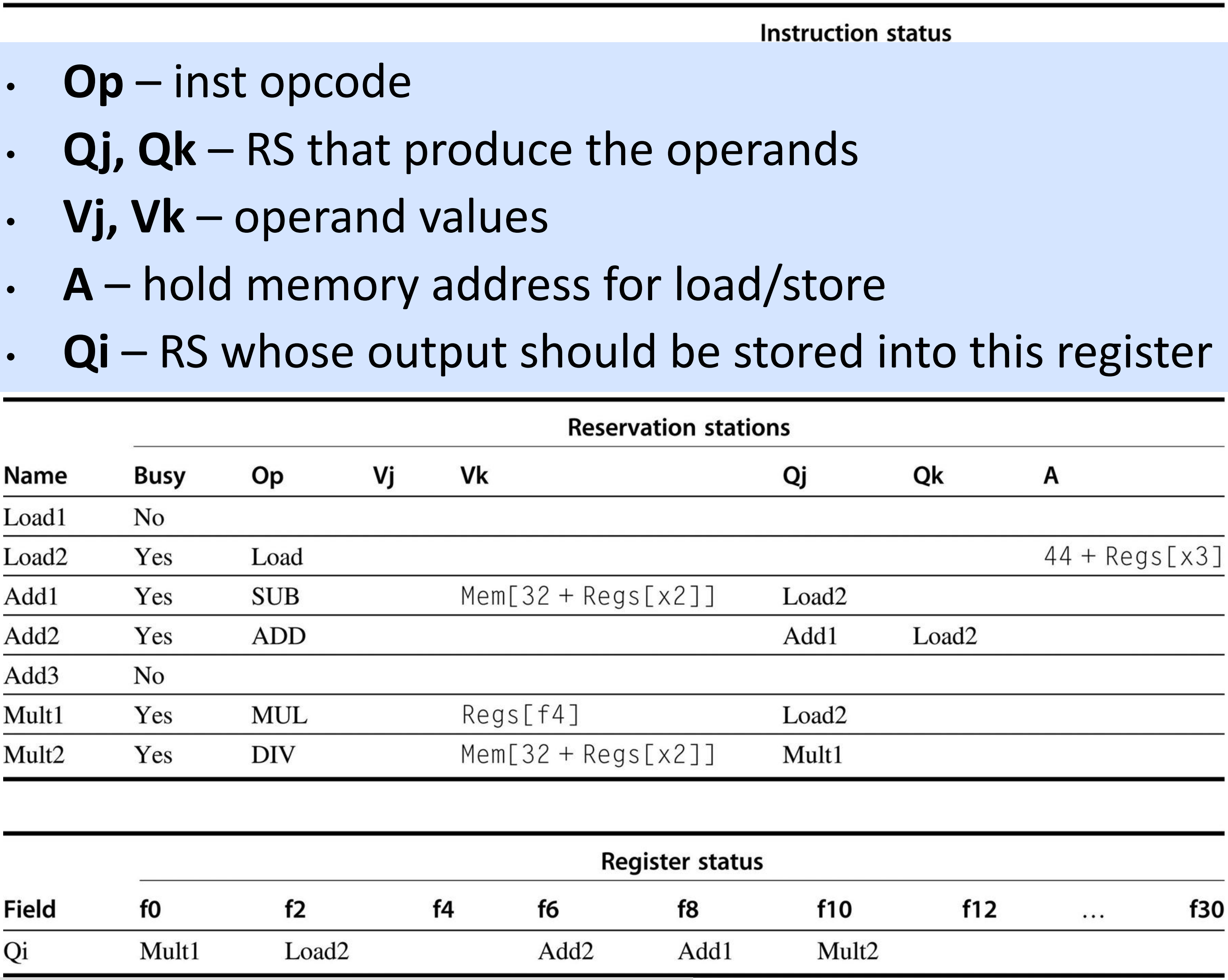
|  | Instruction status |  |  |
|---|---|---|---|
| Instruction | Issue | Execute | Write result |
| `fld f6,32(x2)` | √ | √ | √ |
| `fld f2,44(x3)` | √ | √ |  |
| `fmul.d f0,f2,f4` | √ |  |  |
| `fsub.d f8,f2,f6` | √ |  |  |
| `fdiv.d f0,f0,f6` | √ |  |  |
| `fadd.d f6,f8,f2` | √ |  |  |

|  | Reservation stations |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| Name | Busy | Op | Vj | Vk | Qj | Qk | A |
| Load1 | No |  |  |  |  |  |  |
| Load2 | Yes | Load |  |  |  |  | 44 + Regs[x3] |
| Add1 | Yes | SUB |  | Mem[32 + Regs[x2]] | Load2 |  |  |
| Add2 | Yes | ADD |  |  | Add1 | Load2 |  |
| Add3 | No |  |  |  |  |  |  |
| Mult1 | Yes | MUL |  | Regs[f4] | Load2 |  |  |
| Mult2 | Yes | DIV |  | Mem[32 + Regs[x2]] | Mult1 |  |  |

|  | Register status |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| Field | f0 | f2 | f4 | f6 | f8 | f10 | f12 | . . . | f30 |
| Qi | Mult1 | Load2 |  | Add2 | Add1 | Mult2 |  |  |  |

# Reservation Station Fields

- **Op** – inst opcode
- **Qj, Qk** – RS that produce the operands
- **Vj, Vk** – operand values
- **A** – hold memory address for load/store
- **Qi** – RS whose output should be stored into this register



Instruction status

Reservation stations

| Name | Busy | Op | Vj | Vk | Qj | Qk | A |
|---|---|---|---|---|---|---|---|
| Load1 | No | | | | | | |
| Load2 | Yes | Load | | | | | 44 + Regs[x3] |
| Add1 | Yes | SUB | | Mem[32 + Regs[x2]] | Load2 | | |
| Add2 | Yes | ADD | | | Add1 | Load2 | |
| Add3 | No | | | | | | |
| Mult1 | Yes | MUL | | Regs[f4] | Load2 | | |
| Mult2 | Yes | DIV | | Mem[32 + Regs[x2]] | Mult1 | | |

Register status

| Field | f0 | f2 | f4 | f6 | f8 | f10 | f12 | ... | f30 |
|---|---|---|---|---|---|---|---|---|---|
| Qi | Mult1 | Load2 | | Add2 | Add1 | Mult2 | | | |

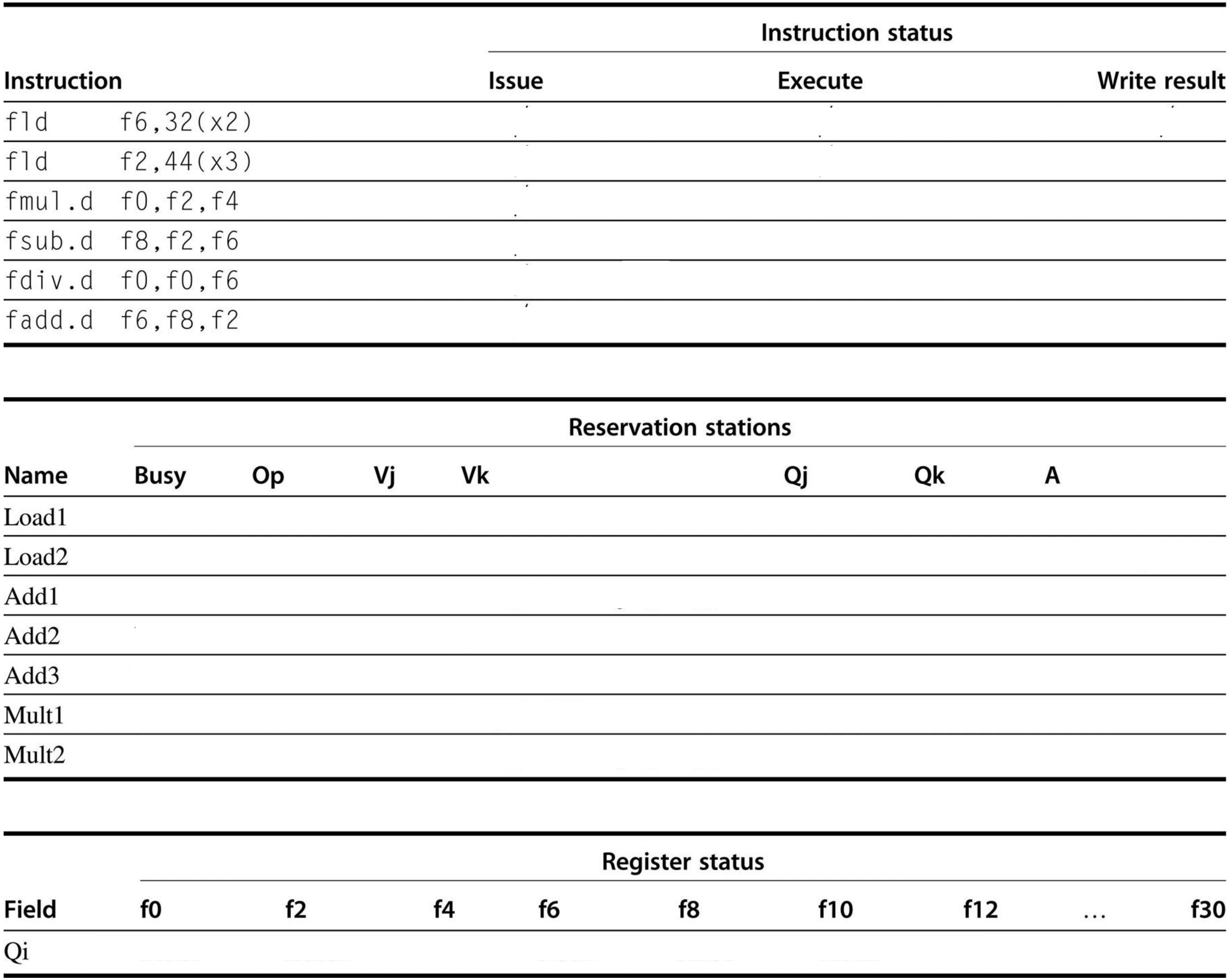
| | Instruction status |
|---|---|
| Instruction | Write result |
| fld f6,32(x2) | |
| fld f2,44(x3) | |
| fmul.d f0,f2,f4 | |
| fsub.d f8,f2,f6 | |
| fdiv.d f0,f0,f6 | |
| fadd.d f6,f8,f2 | |

Latencies of FUs

- Load = 1 cycle
- Add/sub = 2 cycles
- Mult = 6 cycles
- Div = 12 cycles

| | Reservation stations | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | Busy | Op | Vj | Vk | Qj | Qk | A |
| Load1 | | | | | | | |
| Load2 | | | | | | | |
| Add1 | | | | | | | |
| Add2 | | | | | | | |
| Add3 | | | | | | | |
| Mult1 | | | | | | | |
| Mult2 | | | | | | | |

| | Register status | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Field | f0 | f2 | f4 | f6 | f8 | f10 | f12 | … | f30 |
| Qi | | | | | | | | | |



| Instruction | Instruction status | | |
|---|---|---|---|
| | Issue | Execute | Write result |
| `fld f6,32(x2)` | | | |
| `fld f2,44(x3)` | | | |
| `fmul.d f0,f2,f4` | | | |
| `fsub.d f8,f2,f6` | | | |
| `fdiv.d f0,f0,f6` | | | |
| `fadd.d f6,f8,f2` | | | |

| | Reservation stations | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | Busy | Op | Vj | Vk | Qj | Qk | A |
| Load1 | | | | | | | |
| Load2 | | | | | | | |
| Add1 | | | | | | | |
| Add2 | | | | | | | |
| Add3 | | | | | | | |
| Mult1 | | | | | | | |
| Mult2 | | | | | | | |

| | Register status | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Field | f0 | f2 | f4 | f6 | f8 | f10 | f12 | … | f30 |
| Qi | | | | | | | | | |

# A Loop Example

```
Loop: fld       f0, 0(x1)
      fmul      f4, f0, f2
      fsd       f4, 0(x1)
      addi      x1, x1, -8
      bne       x1, x2, Loop
```

| | |
|---|---|
| f#: | FP registers |
| x#: | integer registers |

Assume predict-taken for loop unrolling

| | Instruction status | | | |
|---|---|---|---|---|
| Instruction | From iteration | Issue | Execute | Write result |
| fld f0,0(x1) | 1 | √ | √ | |
| fmul.d f4,f0,f2 | 1 | √ | | |
| fsd f4,0(x1) | 1 | √ | | |
| fld f0,0(x1) | 2 | √ | √ | |
| fmul.d f4,f0,f2 | 2 | √ | | |
| fsd f4,0(x1) | 2 | √ | | |

| | Reservation stations | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | Busy | Op | Vj | Vk | Qj | Qk | A |
| Load1 | Yes | Load | | | | | Regs[x1] + 0 |
| Load2 | Yes | Load | | | | | Regs[x1] – 8 |
| Add1 | No | | | | | | |
| Add2 | No | | | | | | |
| Add3 | No | | | | | | |
| Mult1 | Yes | MUL | | Regs[f2] | Load1 | | |
| Mult2 | Yes | MUL | | Regs[f2] | Load2 | | |
| Store1 | Yes | Store | Regs[x1] | | | Mult1 | |
| Store2 | Yes | Store | Regs[x1] – 8 | | | Mult2 | |

| | Register status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Field | f0 | f2 | f4 | f6 | f8 | f10 | f12 | ... | f30 |
| Qi | Load2 | | Mult2 | | | | | | |

# Ordering of Load and Store

- If load and store accesses different memory, they can be re-ordered
- Otherwise, reordering causes WAW/RAW hazards

```
sd      f1, XXX
ld      f2, XXX
```

```
ld      f1, XXX
sd      f2, XXX
```

```
sd      f1, XXX
sd      f2, XXX
```

- Consider load is next to issue
  - Computer effective address for load
  - Check the A field in all store buffers for match
  - Hold load if a match is found
- Store is handled similarly
  - Check both load and store buffers

# Tomasulo's Algorithm – Summary

- Rename registers to RS
  - To eliminates WAW and WAR hazards
- Buffer RS # for operands currently unavailable
  - To eliminate RAW hazards
- Tolerate unpredictable delays of mem hierarchy
  - Allow instructions to execute while waiting for cache misses
- Achieve high performance w/o compiler optimization
- HW complexity increases substantially
  - Increase area and power

# 3.6 Hardware-Based Speculation

# Hardware-Based Speculation

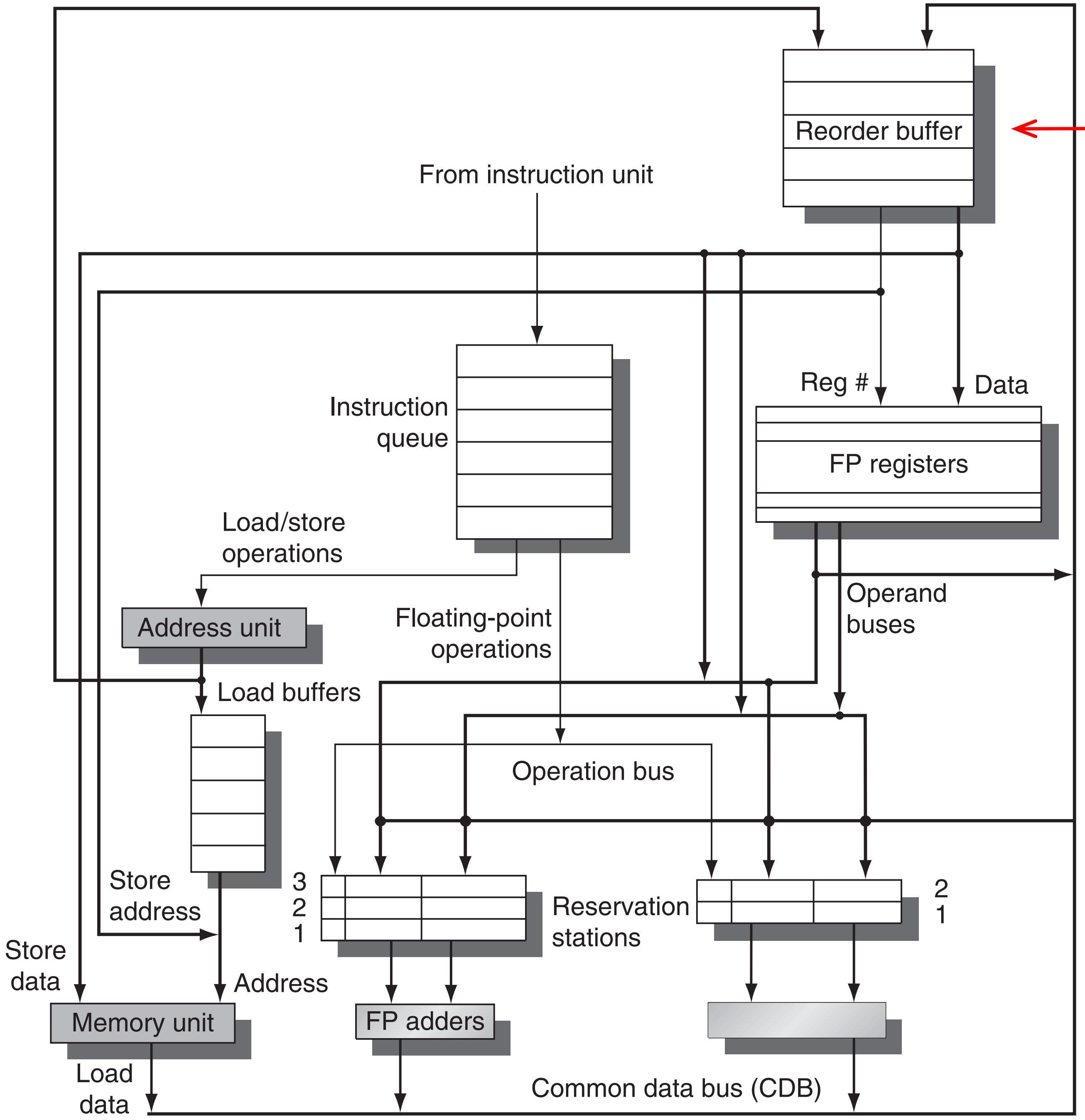
- Combine **dynamic scheduling** and **branch prediction**
- Execute instructions along predicted execution paths but only commit the results if prediction was correct
  - Speculative execution
- Instruction commit: allowing an instruction to update the register/mem when instruction is no longer speculative
  - i.e. branch prediction is correct.
- Instruction execute out-of-order, commit in-order
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - I.e. updating state or taking an execution

# Reorder Buffer

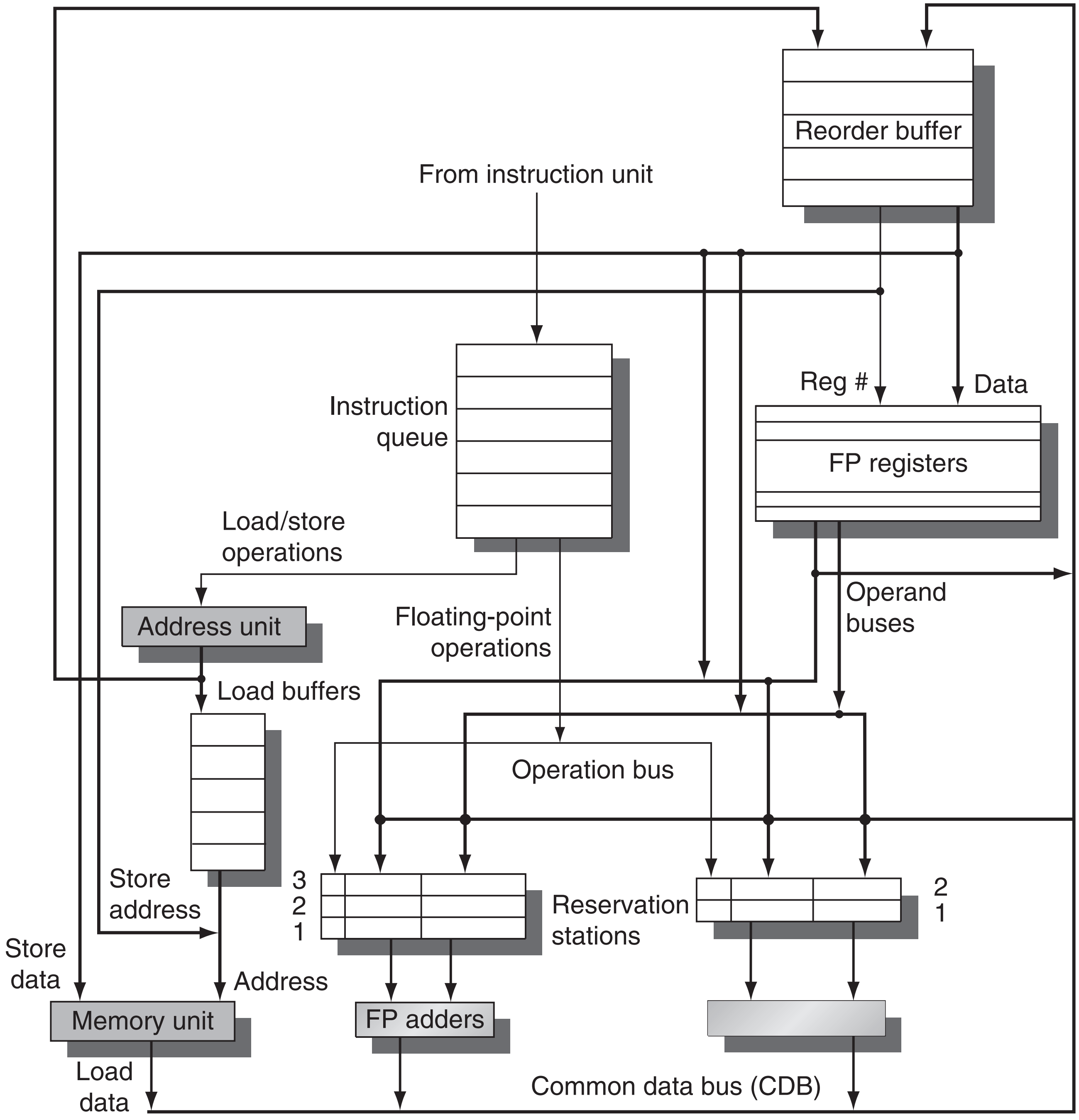
- Reorder buffer (ROB) – holds the result of instruction between completion and commit
- Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number/memory address
  - Value field: output value
  - Ready field: completed execution?
- Source operands are in
  - Registers <– committed instructions
  - ROB <– instructions that complete execution before commt
- Modify reservation stations:
  - Operand sources (Qj or Qk) are now ROB entries instead of RS#

# Reorder Buffer

- ROB and store buffer are merged
- Values to registers/memory are not written until an instruction commits
- On mis-prediction:
  - Speculated entries in ROB are cleared
- Exceptions:
  - Not recognized until it is ready to commit

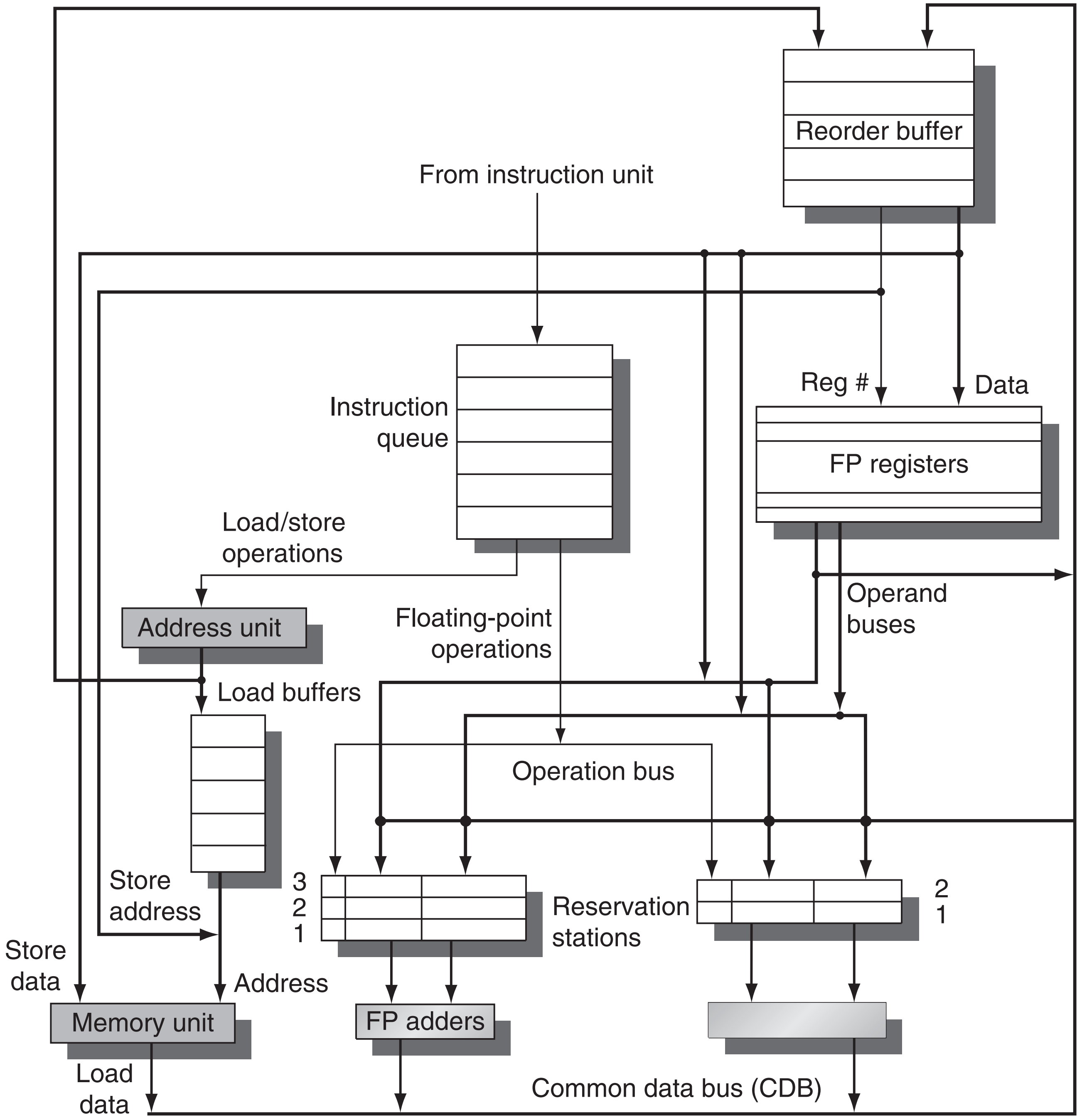


Combined w
store buffers –
A queue



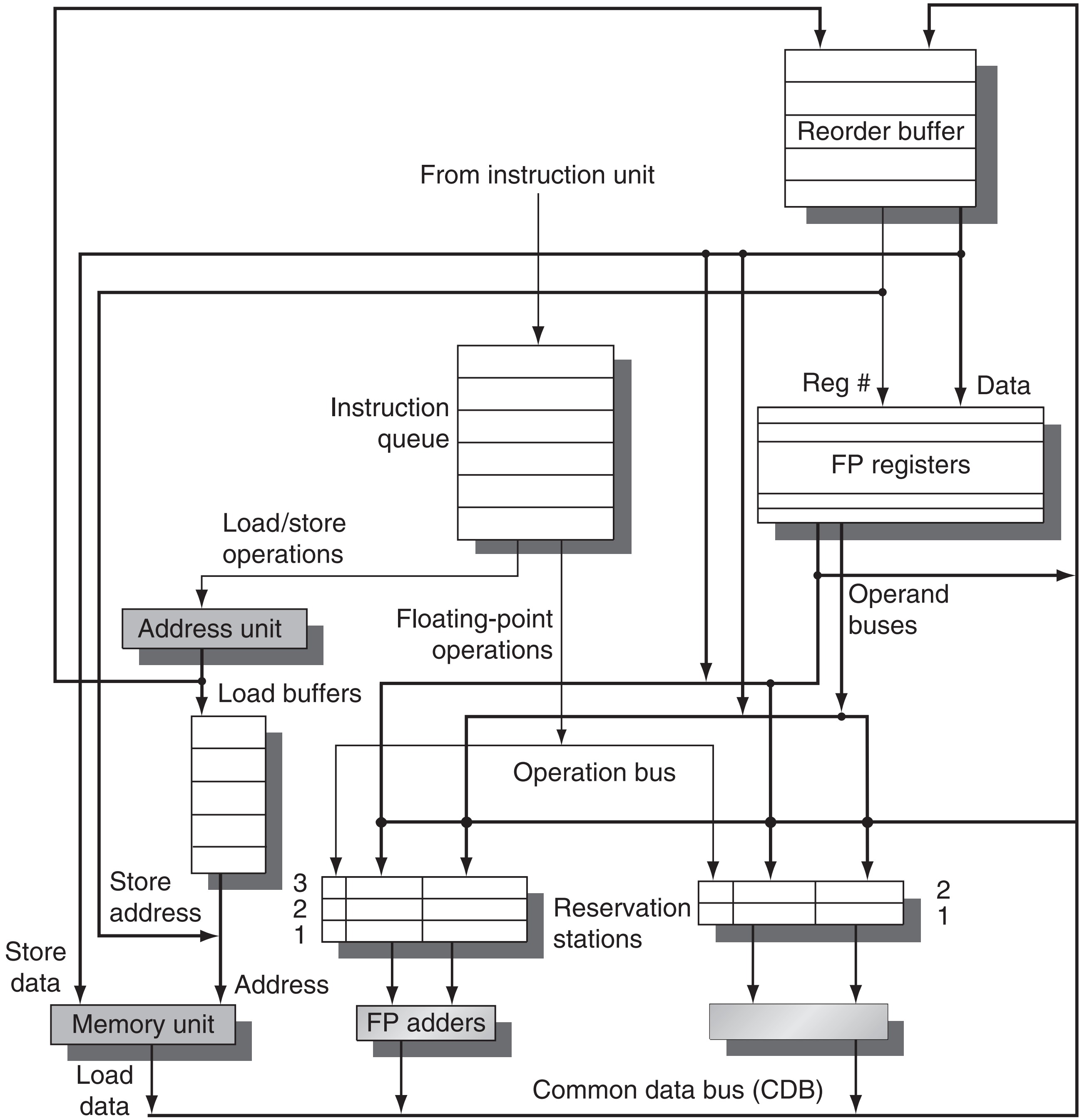
- **Issue** an instruction if there is an empty RS and ROB. Stall otherwise.
- Send the operands to RS if they are available or ROB# if not.
- Tag the instruction result with the ROB# for the destination register

(ROB#, value)



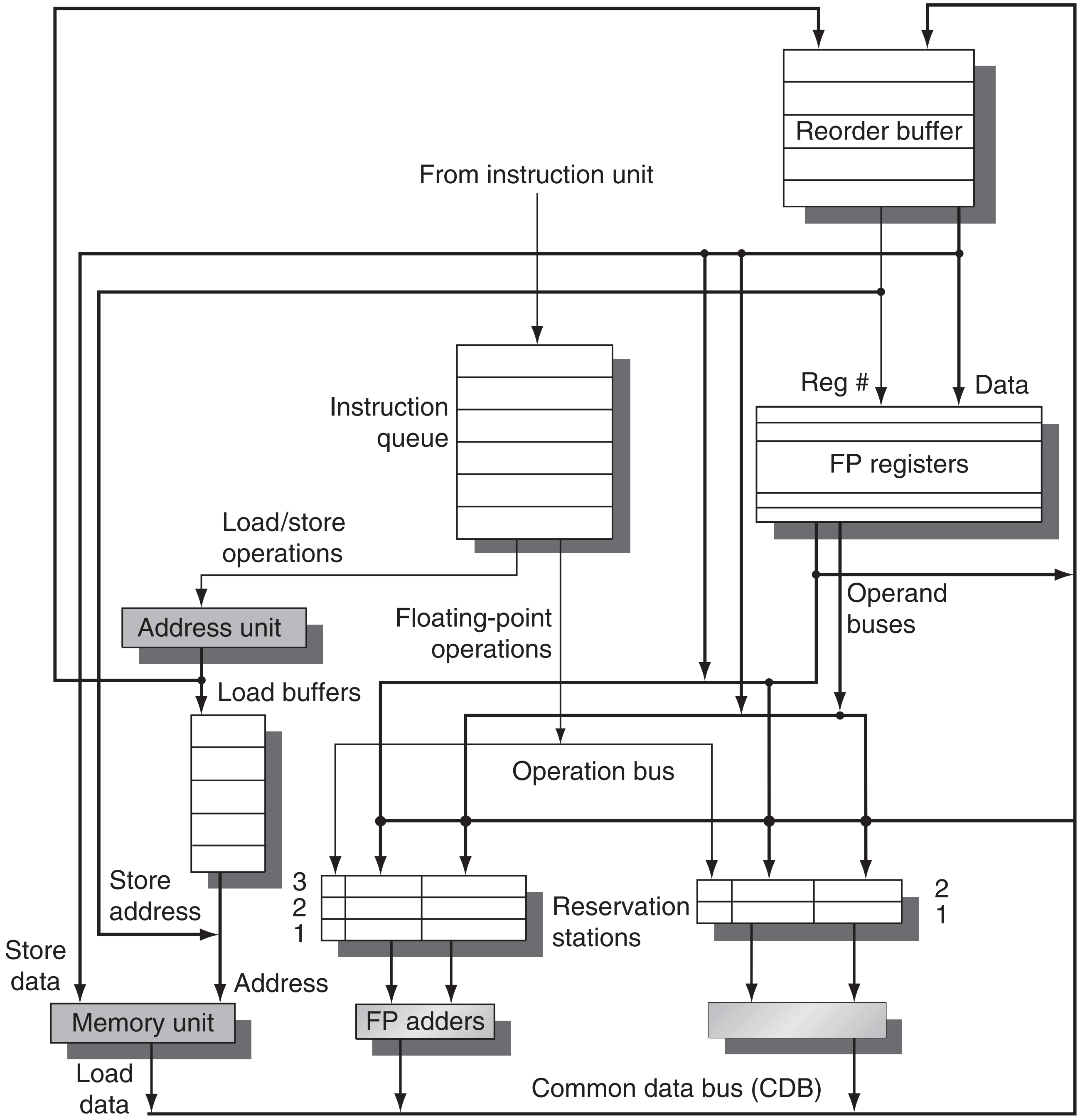
- **Issue**
- **Execute –** start after both operands are available.
- Results are tagged with ROB#

(ROB#, value)



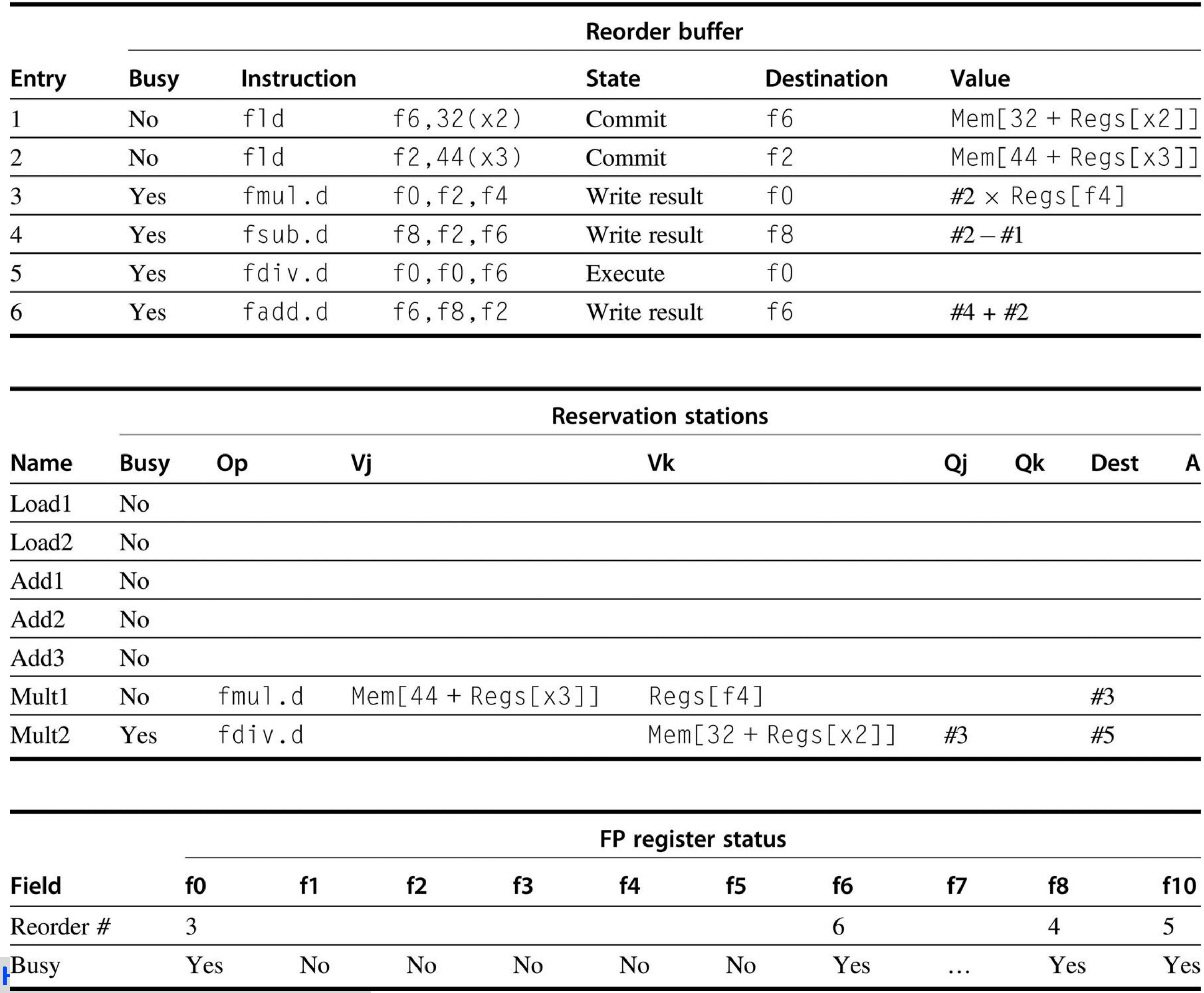
- **Issue**
- **Execute –** start after both operands are available.
- **Write Result –** result put on CDB and goes to ROB and RS
- Any RS or ROB entry expecting a result with ROB# will grab value when <ROB#, value> is put on CDB

(ROB#, value)



- **Issue**
- **Execute**
- **Write Result**
- **Commit –** write result of instruction at the head of ROB
- If the instruction is NOT branch, update M/Reg with the value in ROB
- If the head of ROB is a mis-predicted branch, flush ROB, and restart from the branch target.
- Otherwise, commit as normal.

(ROB#, value)



**Reorder buffer**

| Entry | Busy | Instruction | | State | Destination | Value |
|---|---|---|---|---|---|---|
| 1 | No | fld | f6,32(x2) | Commit | f6 | Mem[32 + Regs[x2]] |
| 2 | No | fld | f2,44(x3) | Commit | f2 | Mem[44 + Regs[x3]] |
| 3 | Yes | fmul.d | f0,f2,f4 | Write result | f0 | #2 × Regs[f4] |
| 4 | Yes | fsub.d | f8,f2,f6 | Write result | f8 | #2 – #1 |
| 5 | Yes | fdiv.d | f0,f0,f6 | Execute | f0 | |
| 6 | Yes | fadd.d | f6,f8,f2 | Write result | f6 | #4 + #2 |

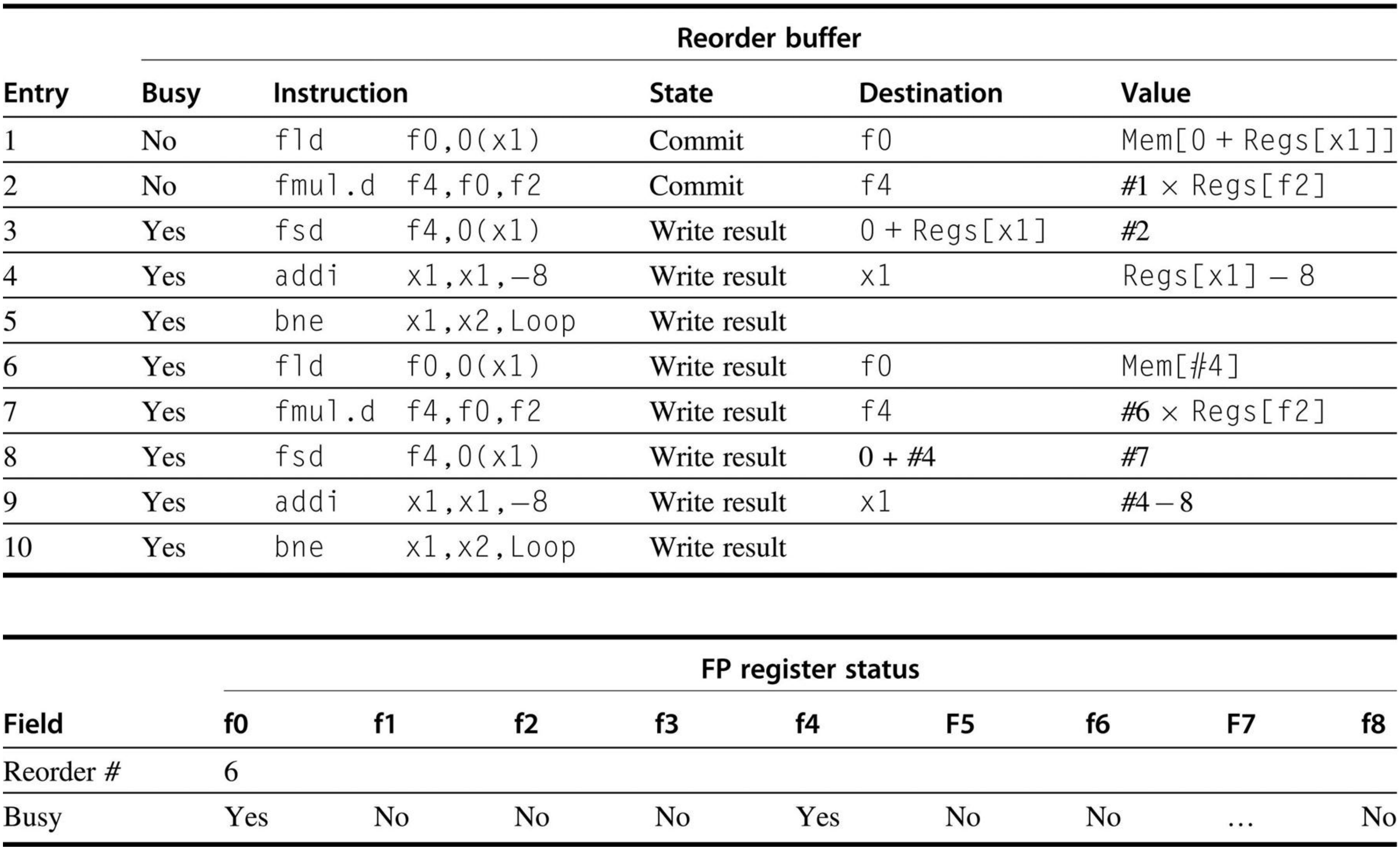
**Reservation stations**

| Name | Busy | Op | Vj | Vk | Qj | Qk | Dest | A |
|---|---|---|---|---|---|---|---|---|
| Load1 | No | | | | | | | |
| Load2 | No | | | | | | | |
| Add1 | No | | | | | | | |
| Add2 | No | | | | | | | |
| Add3 | No | | | | | | | |
| Mult1 | No | fmul.d | Mem[44 + Regs[x3]] | Regs[f4] | | | #3 | |
| Mult2 | Yes | fdiv.d | | Mem[32 + Regs[x2]] | #3 | | #5 | |

**FP register status**

| Field | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | f8 | f10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Reorder # | 3 | | | | | | 6 | | 4 | 5 |
| Busy | Yes | No | No | No | No | No | Yes | … | Yes | Yes |

# A Loop Example



| | Reorder buffer | | | | |
|---|---|---|---|---|---|
| Entry | Busy | Instruction | State | Destination | Value |
| 1 | No | fld f0,0(x1) | Commit | f0 | Mem[0 + Regs[x1]] |
| 2 | No | fmul.d f4,f0,f2 | Commit | f4 | #1 × Regs[f2] |
| 3 | Yes | fsd f4,0(x1) | Write result | 0 + Regs[x1] | #2 |
| 4 | Yes | addi x1,x1,−8 | Write result | x1 | Regs[x1] – 8 |
| 5 | Yes | bne x1,x2,Loop | Write result | | |
| 6 | Yes | fld f0,0(x1) | Write result | f0 | Mem[#4] |
| 7 | Yes | fmul.d f4,f0,f2 | Write result | f4 | #6 × Regs[f2] |
| 8 | Yes | fsd f4,0(x1) | Write result | 0 + #4 | #7 |
| 9 | Yes | addi x1,x1,−8 | Write result | x1 | #4 – 8 |
| 10 | Yes | bne x1,x2,Loop | Write result | | |

| | FP register status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Field | f0 | f1 | f2 | f3 | f4 | F5 | f6 | F7 | f8 |
| Reorder # | 6 | | | | | | | | |
| Busy | Yes | No | No | No | Yes | No | No | … | No |

# Hazards Through Memory

- No WAW/WAR hazards by
  - In-order commit to update reg/memory
- *Store updates MEM in commit, while load reads MEM in execute*
- Avoid RAW hazards through MEM by
  - Not allowing a load to read Mem if its **A** field matches the destination field of any active ROB entry for a store, and
  - Maintaining the program order of effective address computation of a load *wrt* all earlier stores.

```
sd      f1, XXX
ld      f2, XXX
```

# Hardware Speculation – Summary

- Out-of-order execution, in-order commit
- On branch mis-prediction, flush ROB, and processor restart from the branch target.
  - Instructions before the branch are completed as normal
- Branch prediction has higher importance.
  - Impact of mis-prediction is higher
- Instruction exception is delayed until instruction commit
  - Allow precise exception
- WAW/WAR hazards are eliminated by in-order commit
- No RAW hazards with additional control

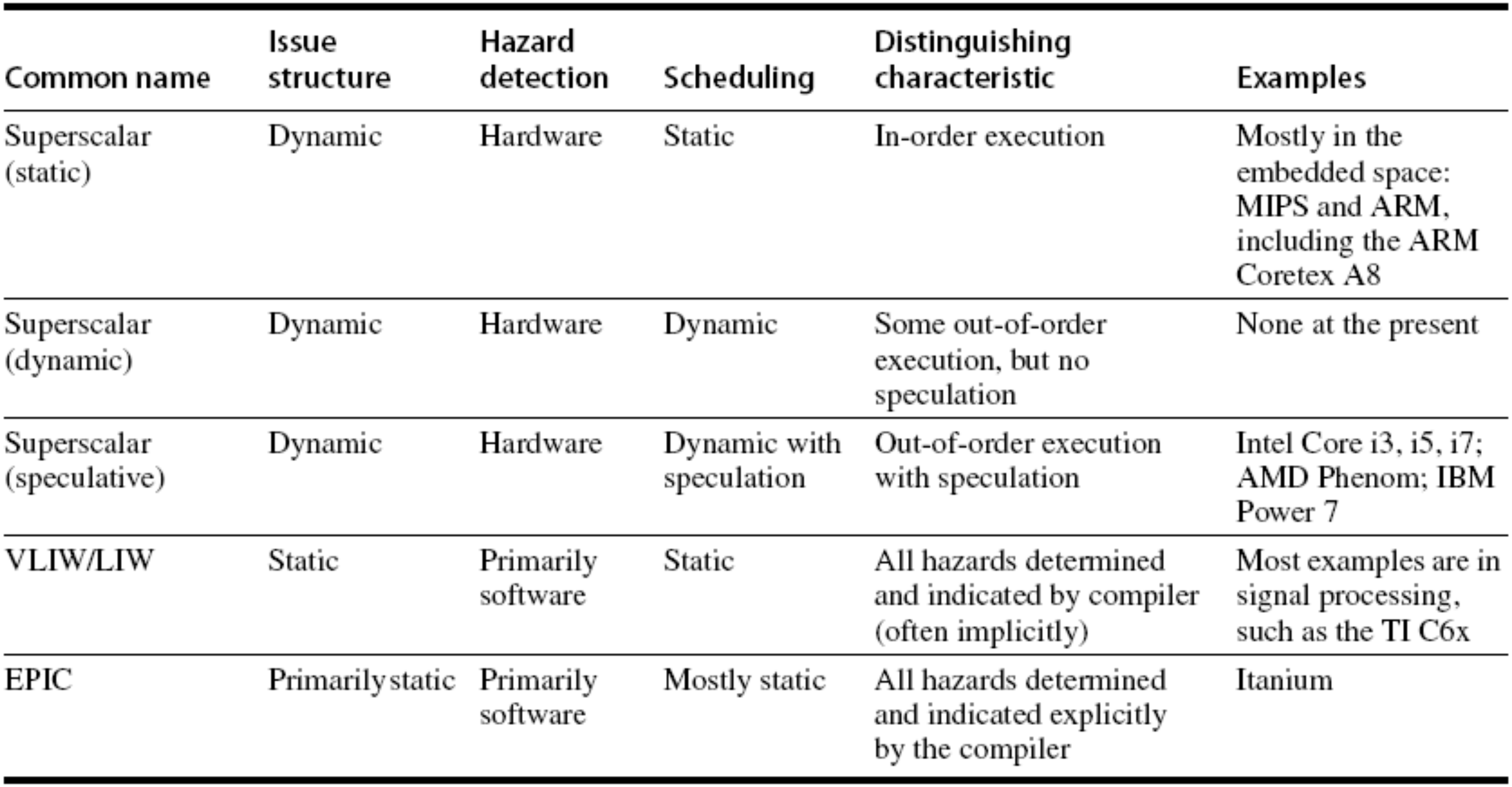
# 3.7 -- 3.8 Multiple Issue

# Multiple Issue and Static Scheduling

- To achieve CPI < 1, need to complete multiple instructions per clock

- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - dynamically scheduled superscalar processors

# Multiple Issue



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

# VLIW Processors

- Package multiple operations into one instruction

- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references

- Must be enough parallelism in code to fill the available slots
  - loop unrolling is a major technique

# VLIW Processors

- Disadvantages:
  - Statically finding parallelism – loop unrolling
  - Single code size increase
  - No hazard detection hardware
  - Binary code compatibility

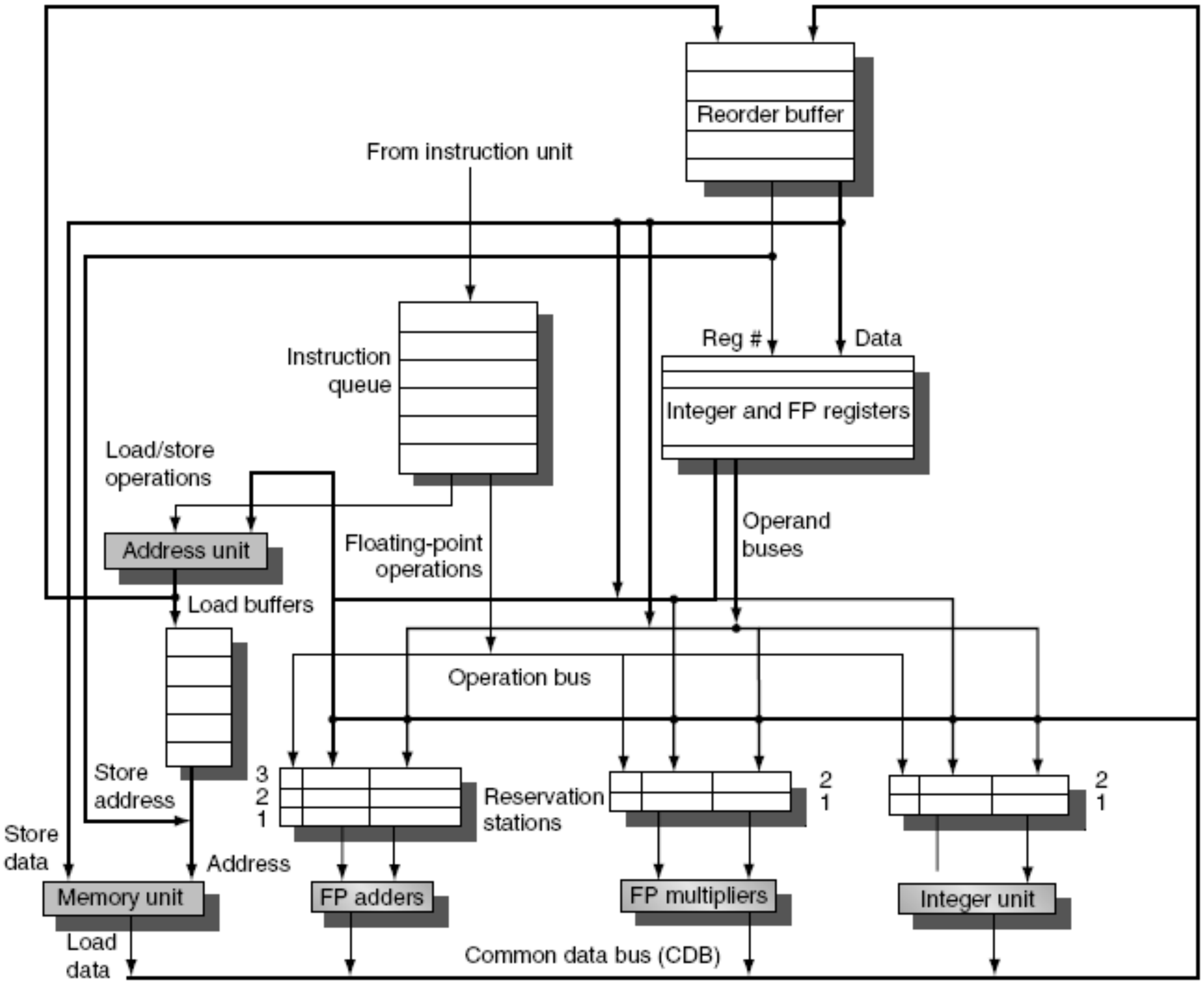
# Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation

- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
  - Hybrid approaches

- Issue logic can become bottleneck
  - see fig. 3.22 for issue logic for (FP load + FP op)

# Overview of Design



Multiple FUs (integer and FP)

An instruction issued for each FU

Instructions issued in-order

RS and ROB updated for all issued instructions

# Multiple Issue

- Limit the number of instructions of a given class that can be issued in a “bundle”, and pre-allocate RS and ROB
  - I.e. on FP, one integer, one load, one store

- Examine all the dependencies among the instructions in the bundle

- If dependencies exist in bundle, encode them in reservation stations

- Also need multiple completion/commit
  - easier – dependencies among issued instructions resolved

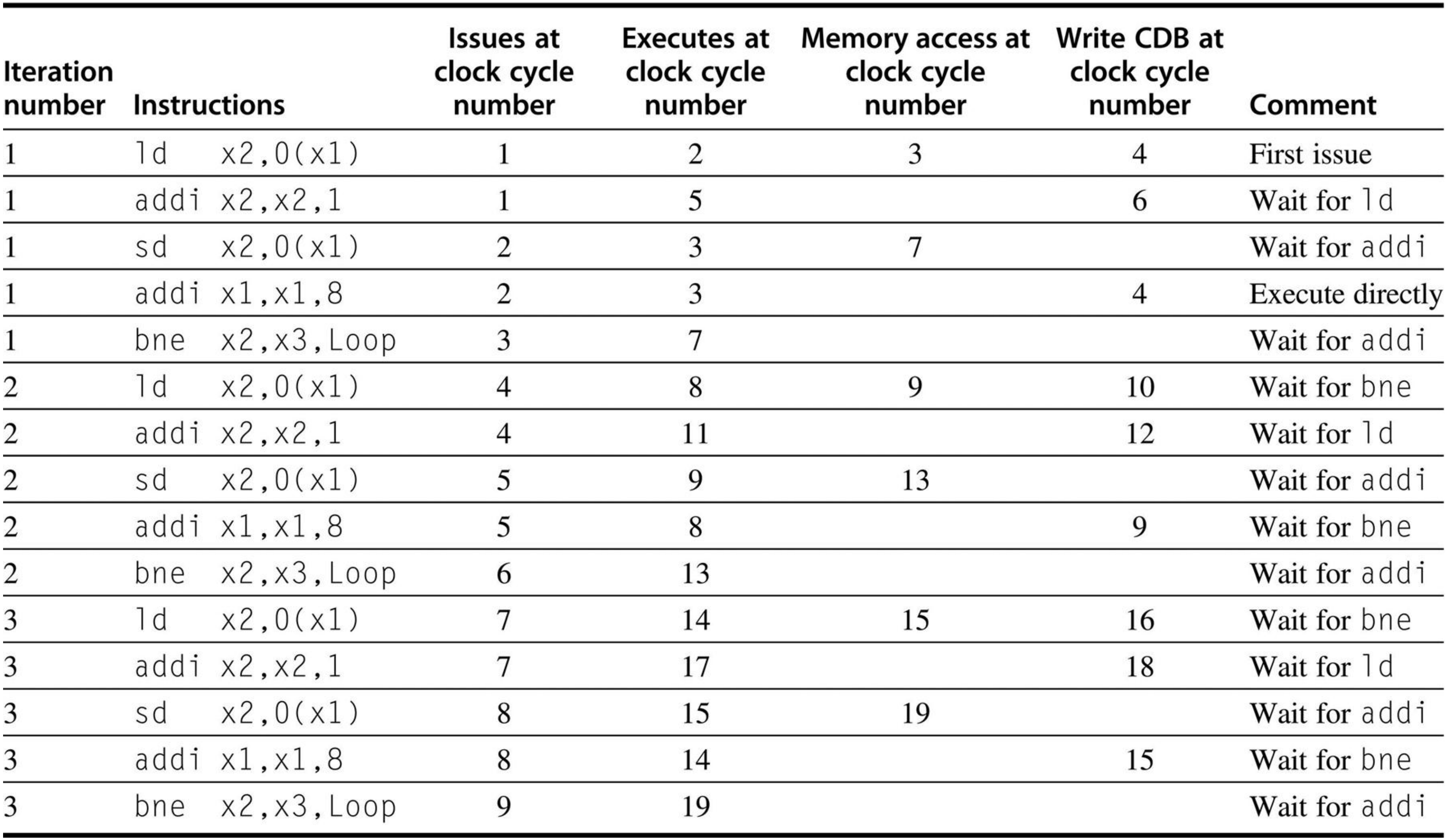
# Example

```
Loop:   ld      x2, 0(x1)               ;R2=array element
        addi    x2, x2, 1               ;increment R2
        sd      x2, 0(x1)               ;store result
        addi    x1, x1, 8               ;increment pointer
        bne     x2, x3, loop            ;branch if not last element
```

What happens when issue both?

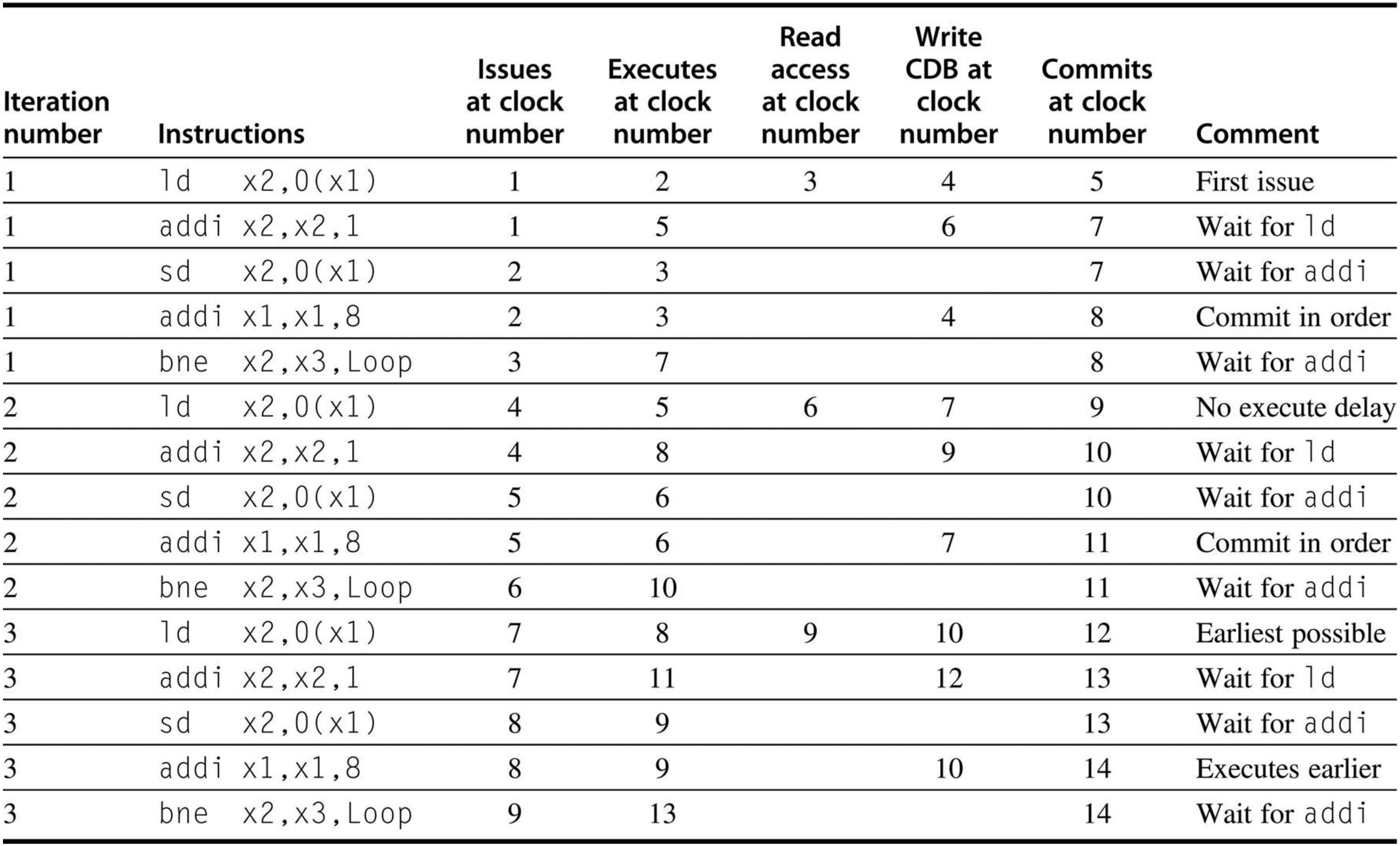
```
        ld      x2, 0(x1)
        addi    x2, x2, 1
```

# Example – Mult-Issue, No Speculation



| Iteration number | Instructions | Issues at clock cycle number | Executes at clock cycle number | Memory access at clock cycle number | Write CDB at clock cycle number | Comment |
|---|---|---|---|---|---|---|
| 1 | ld x2,0(x1) | 1 | 2 | 3 | 4 | First issue |
| 1 | addi x2,x2,1 | 1 | 5 | | 6 | Wait for ld |
| 1 | sd x2,0(x1) | 2 | 3 | 7 | | Wait for addi |
| 1 | addi x1,x1,8 | 2 | 3 | | 4 | Execute directly |
| 1 | bne x2,x3,Loop | 3 | 7 | | | Wait for addi |
| 2 | ld x2,0(x1) | 4 | 8 | 9 | 10 | Wait for bne |
| 2 | addi x2,x2,1 | 4 | 11 | | 12 | Wait for ld |
| 2 | sd x2,0(x1) | 5 | 9 | 13 | | Wait for addi |
| 2 | addi x1,x1,8 | 5 | 8 | | 9 | Wait for bne |
| 2 | bne x2,x3,Loop | 6 | 13 | | | Wait for addi |
| 3 | ld x2,0(x1) | 7 | 14 | 15 | 16 | Wait for bne |
| 3 | addi x2,x2,1 | 7 | 17 | | 18 | Wait for ld |
| 3 | sd x2,0(x1) | 8 | 15 | 19 | | Wait for addi |
| 3 | addi x1,x1,8 | 8 | 14 | | 15 | Wait for bne |
| 3 | bne x2,x3,Loop | 9 | 19 | | | Wait for addi |

# Example – Mult-Issue w Speculation



| Iteration number | Instructions | Issues at clock number | Executes at clock number | Read access at clock number | Write CDB at clock number | Commits at clock number | Comment |
|---|---|---|---|---|---|---|---|
| 1 | `ld x2,0(x1)` | 1 | 2 | 3 | 4 | 5 | First issue |
| 1 | `addi x2,x2,1` | 1 | 5 | | 6 | 7 | Wait for `ld` |
| 1 | `sd x2,0(x1)` | 2 | 3 | | | 7 | Wait for `addi` |
| 1 | `addi x1,x1,8` | 2 | 3 | | 4 | 8 | Commit in order |
| 1 | `bne x2,x3,Loop` | 3 | 7 | | | 8 | Wait for `addi` |
| 2 | `ld x2,0(x1)` | 4 | 5 | 6 | 7 | 9 | No execute delay |
| 2 | `addi x2,x2,1` | 4 | 8 | | 9 | 10 | Wait for `ld` |
| 2 | `sd x2,0(x1)` | 5 | 6 | | | 10 | Wait for `addi` |
| 2 | `addi x1,x1,8` | 5 | 6 | | 7 | 11 | Commit in order |
| 2 | `bne x2,x3,Loop` | 6 | 10 | | | 11 | Wait for `addi` |
| 3 | `ld x2,0(x1)` | 7 | 8 | 9 | 10 | 12 | Earliest possible |
| 3 | `addi x2,x2,1` | 7 | 11 | | 12 | 13 | Wait for `ld` |
| 3 | `sd x2,0(x1)` | 8 | 9 | | | 13 | Wait for `addi` |
| 3 | `addi x1,x1,8` | 8 | 9 | | 10 | 14 | Executes earlier |
| 3 | `bne x2,x3,Loop` | 9 | 13 | | | 14 | Wait for `addi` |

# 3.11 Multithreading

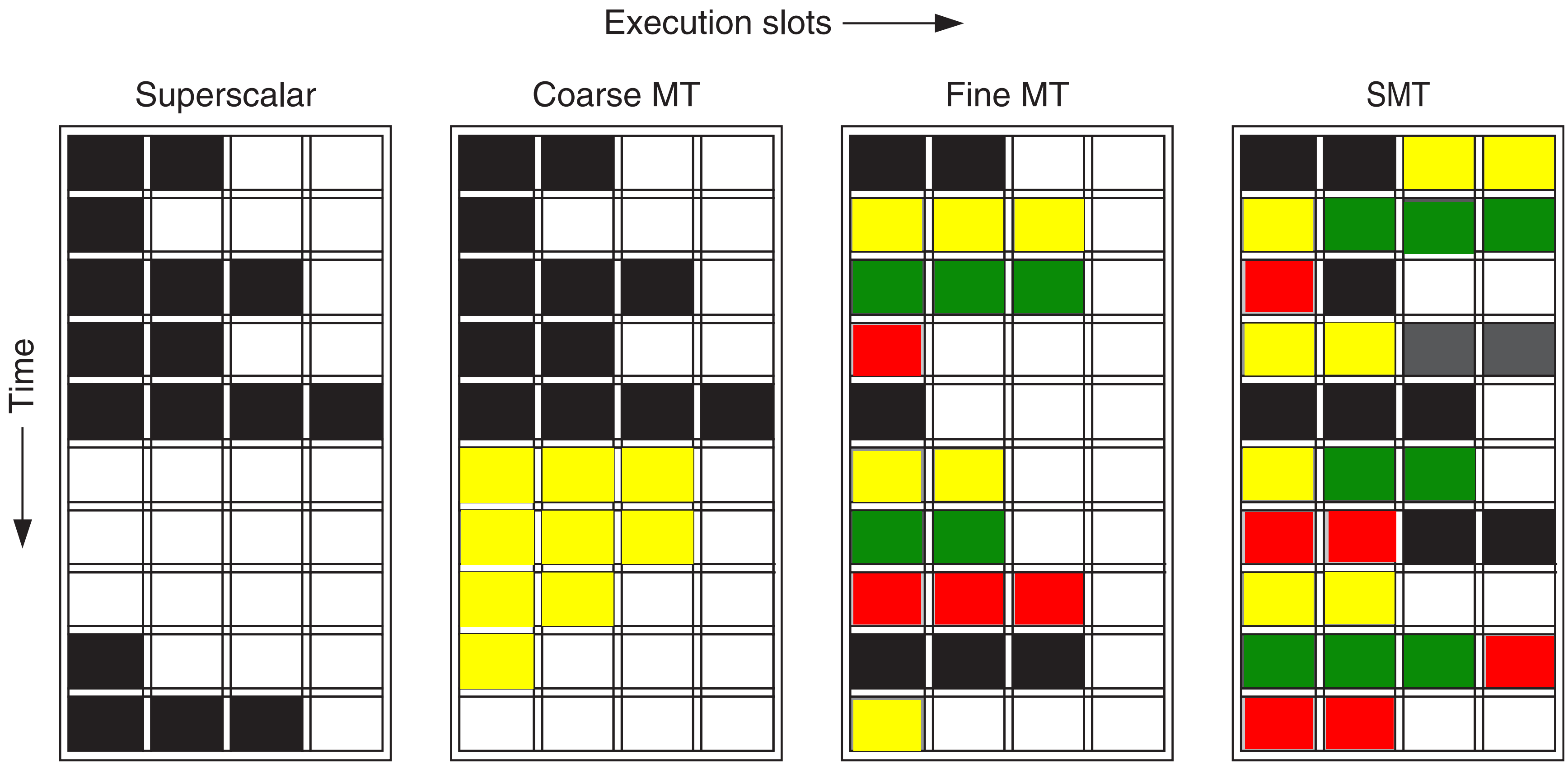
# Limitations of ILP

- Program structure – limited ILP
- WAW/WAR hazards through memory
- Memory bandwidth and latency
  - Pipeline cannot hide latency to access off-chip cache/mem.
  - Remember "memory wall"?
- Impacts of wide issue width
  - Logic complexity ↑
  - Clock rate ↓
  - Power ↑
- Size of ROB and RS

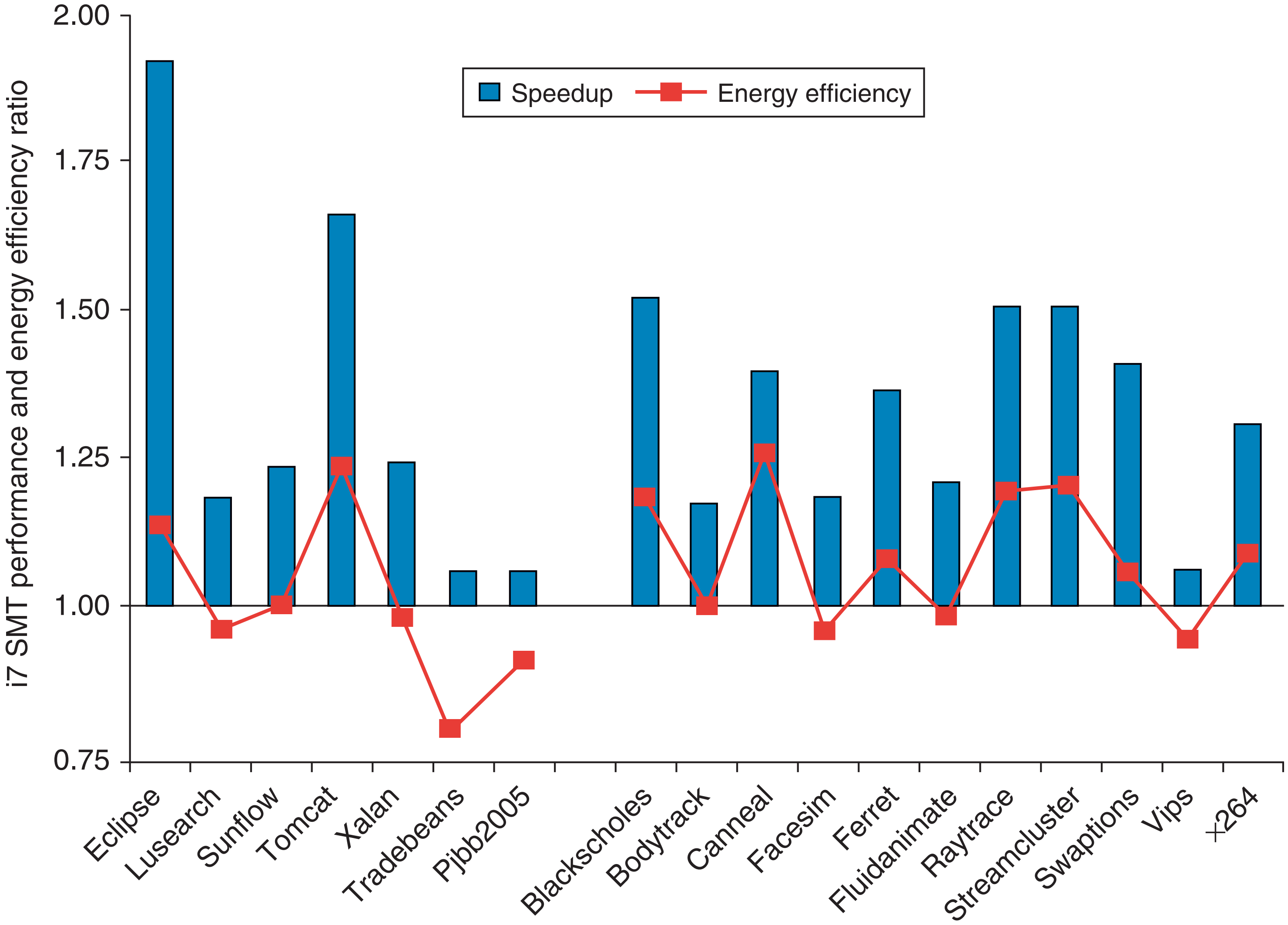
# Multithreading

- Exploiting thread-level parallelism to improve uniprocessor throughput
- Multiprocessing and multithreading in Chapter 5
- Thread-level parallelism (TLP) exists abundantly
  - A property of applications,
  - eg. online transaction processing, SC, web applications,
- Multithreading allows multiple threads to share a processor without process switch
  - Each thread is independently controlled by OS
  - With duplicate private state (reg, PC, etc) for each thread
  - Share FUs and memory, etc

# Multithreading Hardware



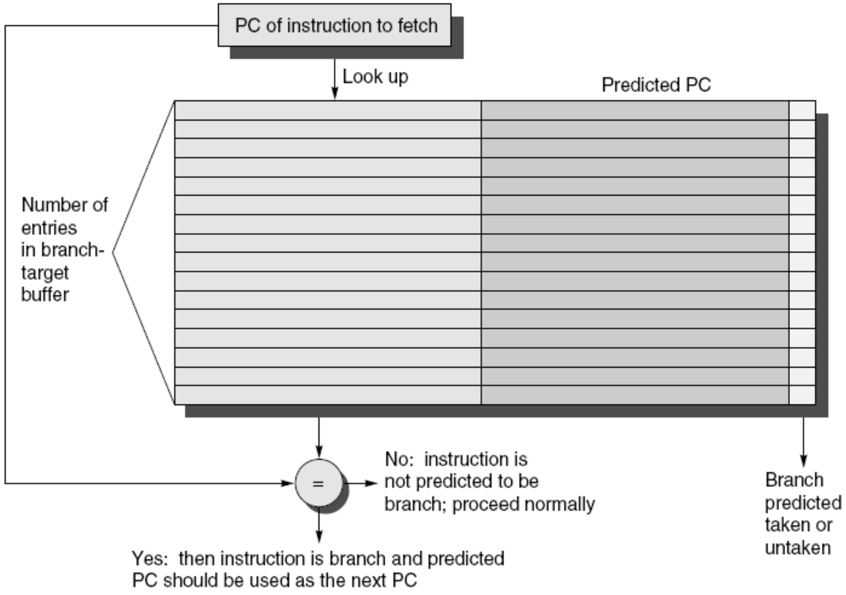
# Effectiveness of SMT

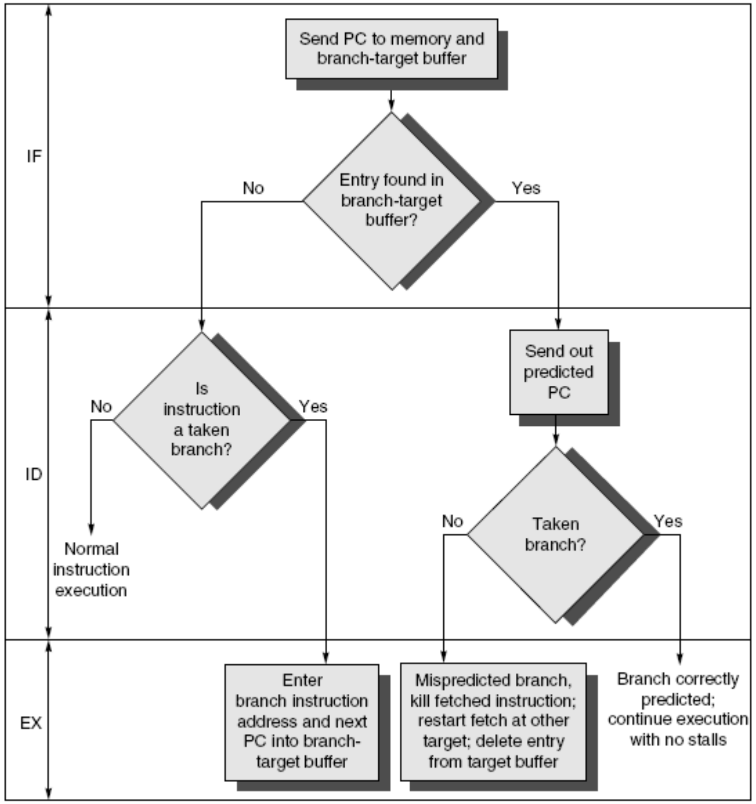


# Backup

# Branch-Target Buffer

- → Need high instruction bandwidth!
  - → Branch-Target buffers
    - → Next PC prediction buffer, indexed by current PC


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken


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

# Branch Folding

- Optimization:
  - Larger branch-target buffer
  - Add target instruction into buffer to deal with longer decoding time required by larger buffer
  - “Branch folding”

# Return Address Predictor

- Most unconditional branches come from function returns
- The same procedure can be called from multiple sites
  - Causes the buffer to potentially forget about the return address from previous calls
- Create return address buffer organized as a stack

# Integrated Instruction Fetch Unit

- Design monolithic unit that performs:
  - Branch prediction
  - Instruction prefetch
    - Fetch ahead
  - Instruction memory access and buffering
    - Deal with crossing cache lines

# Register Renaming

- Register renaming vs. reorder buffers
  - Instead of virtual registers from reservation stations and reorder buffer, create a single register pool
    - Contains visible registers and virtual registers
  - Use hardware-based map to rename registers during issue
  - WAW and WAR hazards are avoided
  - Speculation recovery occurs by copying during commit
  - Still need a ROB-like queue to update table in order
  - Simplifies commit:
    - Record that mapping between architectural register and physical register is no longer speculative
    - Free up physical register used to hold older value
    - In other words:  SWAP physical registers on commit
  - Physical register de-allocation is more difficult

# Integrated Issue and Renaming

- Combining instruction issue with register renaming:
  - Issue logic pre-reserves enough physical registers for the bundle (fixed number?)
  - Issue logic finds dependencies within bundle, maps registers as necessary
  - Issue logic finds dependencies between current bundle and already in-flight bundles, maps registers as necessary

# How Much?

- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)
- Speculating through multiple branches
  - Complicates speculation recovery
  - No processor can resolve multiple branches per cycle

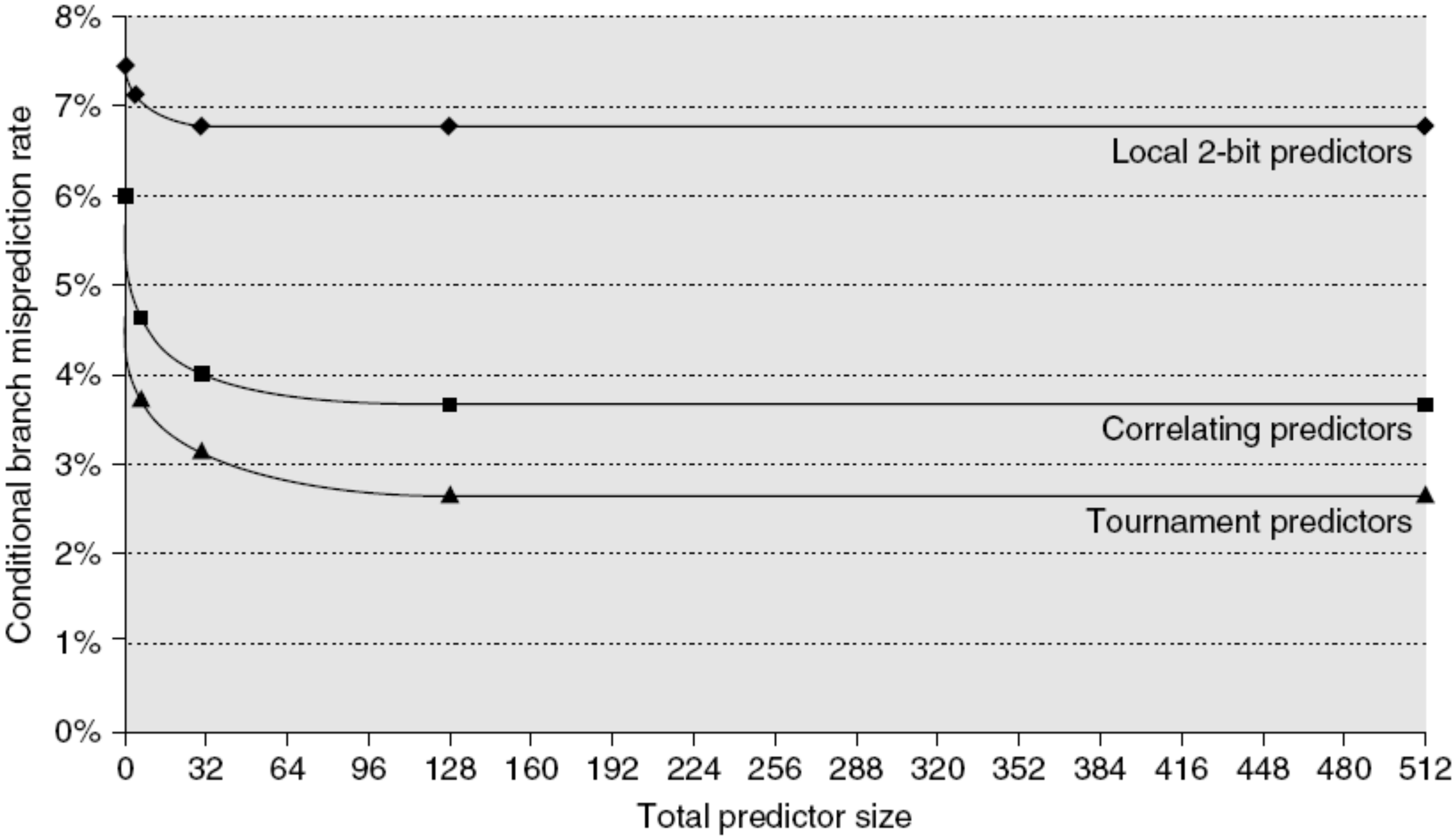
# Energy Efficiency

- Speculation and energy efficiency
  - Note:  speculation is only energy efficient when it significantly improves performance
- Value prediction
  - Uses:
    - Loads that load from a constant pool
    - Instruction that produces a value from a small set of values
  - Not been incorporated into modern processors
  - Similar idea--*address aliasing prediction*--is used on some processors

# Branch Prediction

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding *n* branches
- Local predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- Tournament predictor:
  - Combine correlating predictor with local predictor

# Branch Prediction Performance



Branch predictor performance