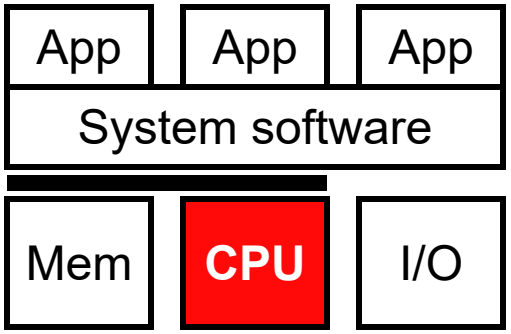
Computer Architecture

Lec 11: Static & Dynamic Scheduling

This Unit: Static & Dynamic Scheduling



- Code scheduling
  - To reduce pipeline stalls
  - To increase ILP (insn level parallelism)
- Static scheduling by the compiler
  - Approach & limitations
- Dynamic scheduling in hardware
  - Register renaming
  - Instruction selection
  - Handling memory operations

Review: Dependences and Hazards

- **Dependence**: relationship between two insns
  - **Data**: two insns use same storage location
  - **Control**: one insn affects whether another executes at all
  - Not a bad thing, programs would be boring without them
  - Enforced by making older insn go before younger one
    - Happens naturally in single-/multi-cycle designs
    - But not in a pipeline

- **Hazard**: dependence & possibility of wrong insn order
  - Effects of wrong insn order must not be externally visible
    - **Stall**: for order by keeping younger insn in same stage
  - Hazards are a bad thing: stalls reduce performance

Review: Data Dependences

- **RAW** (Read After Write) = "true dependence" (true)

  mul r0 * r1 ➔ **r2**

  …

  add **r2** + r3 ➔ r4

- **WAW** (Write After Write) = "output dependence" (false)

  mul r0 * r1➔ **r2**

  …

  add r1 + r3 ➔ **r2**

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

  mul r0 * **r1** ➔ r2

  …

  add r3 + r4 ➔ **r1**

Can We Do Better?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL  R3 ← R1, R2
ADD   R3 ← R3, R1
ADD   R4 ← R6, R7
IMUL  R5 ← R6, R8
ADD   R7 ← R9, R9
```

```
LD    R3 ← R1 (0)
ADD   R3 ← R3, R1
ADD   R4 ← R6, R7
IMUL  R5 ← R6, R8
ADD   R7 ← R9, R9
```

- Answer: First ADD stalls the whole pipeline!
  - ADD cannot dispatch because its source registers unavailable
  - Later **independent** instructions cannot get executed

Code Scheduling

- Scheduling: act of finding independent instructions
  - "Static" done at compile time by the compiler (software)
  - "Dynamic" done at runtime by the processor (hardware)

```
ld [sp+4]➜r2
ld [sp+8]➜r3
add r2,r3➜r1
st r1➜[sp+0]
ld [sp+16]➜r5
ld [sp+20]➜r6
sub r6,r5➜r4
st r4➜[sp+12]
```

Static Scheduling by Compilier

Compiler Scheduling

- Compiler can schedule (move) instructions to reduce stalls
  - Data dependency will not lead to stalls if the insts are far enough
  - **Basic pipeline scheduling**: eliminate back-to-back load-use pairs

Before

```
ld [sp+4]➜r2
ld [sp+8]➜r3
add r2,r3➜r1 //stall
st r1➜[sp+0]
ld [sp+16]➜r5
ld [sp+20]➜r6
sub r6,r5➜r4 //stall
st r4➜[sp+12]
```

After

```
ld [sp+4]➜r2
ld [sp+8]➜r3
ld [sp+16]➜r5
add r2,r3➜r1 //no stall
ld [sp+20]➜r6
st r1➜[sp+0]
sub r6,r5➜r4 //no stall
st r4➜[sp+12]
```

Loop Unrolling

- **Example:**

```
for (j=1; j<=1000; j++) x[j]=x[j]+s;

Loop: l.d    f0, 0(r1)    ;f0=vector element
      add.d  f4, f0, f2   ;add scalar from f2
      s.d    f4, 0(r1)    ;store result
      subi   r1, r1, 8    ;decrement pointer 8B (DW)
      bnez   r1, Loop     ;branch R1!=zero
      nop                 ;delayed branch slot
```

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
Integer op	Integer op	0

Before Unrolling

```
1 Loop:  l.d     f0,0(r1)    ;f0=vector element
2        stall
3        add.d   f4,f0,f2    ;add scalar in f2
4        stall
5        stall
6        s.d     f4,0(r1)    ;store result
7        subi    r1,r1,8     ;decrement pointer 8B (DW)
8        bnez    r1,Loop     ;branch r1!=zero
9        stall                ;delayed branch slot
```

9 cycles

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
Integer op	Integer op	0

Before Unrolling but with Reordering

```
1 Loop:  l.d     f0,0(r1)
2        stall
3        add.d   f4,f0,f2
4        subi    r1,r1,8
5        bnez    r1,Loop      ;delayed branch
6        s.d     f4,8(r1)     ;altered when move past subi
```

6 cycles

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

Loop Unrolling (4 times)

- Suppose r1%4 = 0

```
Loop:  l.d     f0, 0(r1)
       add.d   f4, f0, f2
       s.d     f4,0(r1)          ;drop subi & bnez
       l.d     f0, -8(r1)
       add.d   f4, f0, f2
       s.d     f4,-8(r1)         ;drop subi & bnez
       l.d     f0, -16(r1)
       add.d   f4, f0, f2
       s.d     f4,-16(r1)        ;drop subi & bnez
       l.d     f0, -24(r1)
       add.d   f4, f0, f2
       s.d     f4,-24(r1)
       subi    r1, r1, #32       ;alter to 4*8
       bnez    r1, LOOP
       nop
```

Loop Unrolling (register renaming)

- register renaming eliminates WAW and WAR dependencies

```
1 Loop: l.d     f0, 0(r1)
2       add.d   f4, f0, f2
3       s.d     0(r1), f4
4       l.d     f0, -8(r1)
5       add.d   f4, f0, f2
6       s.d     -8(r1), f4
7       l.d     f0, -16(r1)
8       add.d   f4, f0, f2
9       s.d     -16(r1), f4
10      l.d     f0, -24(r1)
11      add.d   f4, f0, f2
12      s.d     -24(r1), f4
13      subi    r1, r1, #32
14      bnez    r1, LOOP
15      nop
```

```
1 Loop: l.d     f0, 0(r1)
2       add.d   f4, f0, f2
3       s.d     0(r1), f4
4       l.d     f6, -8(r1)
5       add.d   f8, f6, f2
6       s.d     -8(r1), f8
7       l.d     f10, -16(r1)
8       add.d   f12, f10, f2
9       s.d     -16(r1), f12
10      l.d     f14, -24(r1)
11      add.d   f16, f14, f2
12      s.d     -24(r1), f16
13      subi    r1, r1, #32
14      bnez    r1, LOOP
15      nop
```

27 cycles in four loops, each loop takes 6.8 cycles

Loop Unrolling + Reordering

14 cycles in four loops, each loop takes 3.5 cycles

```
1 Loop: l.d      f0, 0(r1)
2       l.d      f6, -8(r1)
3       l.d      f10, -16(r1)
4       l.d      f14, -24(r1)
5       add.d    f4, f0, f2
6       add.d    f8, f6, f2
7       add.d    f12, f10, f2
8       add.d    f16, f14, f2
9       s.d      0(r1), f4
10      s.d      -8(r1), f8
11      s.d      -16(r1), f12
12      subi     r1, r1, #32
13      bnez     r1, LOOP
14      s.d      8(r1), f16      ;8-32 = -24
```

Compiler Scheduling Requires

- **Large scheduling scope**
  - Independent instruction to put between load-use pairs
  - \+ Original example: large scope, two independent computations
  - – This example: small scope, one computation

Before

```
ld [sp+4]➜r2
ld [sp+8]➜r3
add r2,r3➜r1 //stall
st r1➜[sp+0]
```

After (same!)

```
ld [sp+4]➜r2
ld [sp+8]➜r3
add r2,r3➜r1 //stall
st r1➜[sp+0]
```

- Compiler can create **larger** scheduling scopes
  - For example: loop unrolling & function inlining

Compiler Scheduling Requires

- **Enough registers**
  - To hold additional "live" values
  - Example code contains 7 different values (including **sp**)
  - Before: max 3 values live at any time → 3 registers enough
  - After: max 4 values live → 3 registers not enough

Original

```
ld [sp+4]➜r2
ld [sp+8]➜r1
add r1,r2➜r1 //stall
st r1➜[sp+0]
ld [sp+16]➜r2
ld [sp+20]➜r1
sub r2,r1➜r1 //stall
st r1➜[sp+12]
```

Wrong!

```
ld [sp+4]➜r2
ld [sp+8]➜r1
ld [sp+16]➜r2
add r1,r2➜r1   // wrong r2
ld [sp+20]➜r1
st r1➜[sp+0]   // wrong r1
sub r2,r1➜r1
st r1➜[sp+12]
```

Compiler Scheduling Requires

- **Alias analysis**
  - Ability to tell whether load/store reference same memory locations
    - Effectively, whether load/store can be rearranged
  - Previous example: easy, loads/stores use same base register (**`sp`**)
  - New example: can compiler tell that **`r8`** != **`r9`**?
  - Must be **conservative**

<u>Before</u>

```
ld [r9+4]➜r2
ld [r9+8]➜r3
add r3,r2➜r1 //stall
st r1➜[r9+0]
ld [r8+0]➜r5
ld [r8+4]➜r6
sub r5,r6➜r4 //stall
st r4➜[r8+8]
```

<u>Wrong(?)</u>

```
ld [r9+4]➜r2
ld [r9+8]➜r3
ld [r8+0]➜r5 //does r8==r9?
add r3,r2➜r1
ld [r8+4]➜r6 //does r8+4==r9?
st r1➜[r9+0]
sub r5,r6➜r4
st r4➜[r8+8]
```

Scheduling Scope Limited by Branches

r1 and r2 are inputs

```
loop:
   jz r1, not_found
   ld [r1+0]➔r3
   sub r2,r3➔r4
   jz r4, found
   ld [r1+4]➔r1
   jmp loop
```

Aside: what does this code do?

Legal to move load up past branch?

A Good Case: Static Scheduling of SAXPY

- **SAXPY** (Single-precision A X Plus Y)
  - Linear algebra routine (used in solving systems of equations)

```
for (i=0;i<N;i++)
  Z[i]=(A*X[i])+Y[i];
```

```
0: ldf [X+r1]➜f1        // loop
1: mulf f0,f1➜f2        // A in f0
2: ldf [Y+r1]➜f3        // X,Y,Z are constant addresses
3: addf f2,f3➜f4
4: stf f4➜[Z+r1]
5: addi r1,4➜r1         // i in r1
6: blt r1,r2,0          // N*4 in r2
```

- Static scheduling works great for SAXPY
  - All loop iterations independent
  - Use loop unrolling to increase scheduling scope
  - Aliasing analysis is tractable (just ensure X, Y, Z are independent)
  - Still limited by number of registers

Unrolling & Scheduling SAXPY

- Fuse two (in general, K) iterations of loop
  - Fuse loop control: induction variable (`i`) increment + branch
  - Adjust register names & induction uses (constants → constants+4)
  - Reorder operations to reduce stalls

```
ldf [X+r1]➜f1
mulf f0,f1➜f2
ldf [Y+r1]➜f3
addf f2,f3➜f4
stf f4➜[Z+r1]
addi r1,4➜r1
blt r1,r2,0
ldf [X+r1]➜f1
mulf f0,f1➜f2
ldf [Y+r1]➜f3
addf f2,f3➜f4
stf f4➜[Z+r1]
addi r1,4➜r1
blt r1,r2,0
```

→

```
ldf [X+r1]➜f1
mulf f0,f1➜f2
ldf [Y+r1]➜f3
addf f2,f3➜f4
stf f4➜[Z+r1]

ldf [X+r1+4]➜f5
mulf f0,f5➜f6
ldf [Y+r1+4]➜f7
addf f6,f7➜f8
stf f8➜[Z+r1+4]
addi r1,8➜r1
blt r1,r2,0
```

→

```
ldf [X+r1]➜f1
ldf [X+r2+4]➜f5
mulf f0,f1➜f2
mulf f0,f5➜f6
ldf [Y+r1]➜f3
ldf [Y+r1+4]➜f7
addf f2,f3➜f4
addf f6,f7➜f8
stf f4➜[Z+r1]
stf f8➜[Z+r1+4]
addi r1,8➜r1
blt r1,r2,0
```

Compiler Scheduling Limitations

- Scheduling scope
  - Example: can't generally move memory operations past branches
- Limited number of registers (set by ISA)
- Inexact "memory aliasing" information
  - Often prevents reordering of loads above stores by compiler
- Caches misses (or any runtime event) confound scheduling
  - How can the compiler know which loads will miss vs hit?
  - Can impact the compiler's scheduling decisions

Can Hardware Overcome These Limits?

- **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, appearance of in-order
- Increases scheduling scope
  - Does loop unrolling transparently!
  - Uses branch prediction to "unroll" branches
- Examples:
  - Pentium Pro/II/III (3-wide), Core 2 (4-wide),
    Alpha 21264 (4-wide), MIPS R10000 (4-wide), Power5 (5-wide)

Dynamic Scheduling by Hardware

Dynamic Scheduling

- Also called "**Out of Order Execution**"
  - Done by the hardware on-the-fly during execution
- Idea: Move the dependent instructions out of the way of independent ones (s.t. independent ones can execute)
  - Rest areas for dependent instructions: **Reservation stations**
- Monitor the source "values" of each instruction in the resting area
- When all source "values" of an instruction are available, "fire" (i.e. dispatch) the instruction
  - Instructions dispatched in dataflow (not control-flow) order
- Benefit:
  - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long-latency operation

Enabling OoO Execution

1. Need to link the consumer of a value to the producer
   - Register renaming: Associate a "tag" with each data value
2. Need to buffer instructions until they are ready to execute
   - Insert instruction into reservation stations after renaming
3. Instructions need to keep track of readiness of source values
   - Broadcast the "tag" when the value is produced
   - Instructions compare their "source tags" to the broadcast tag → if match, source value becomes ready
4. When all source values of an instruction are ready, need to dispatch the instruction to its functional unit (FU)
   - Instruction wakes up if all sources are ready
   - If multiple instructions are awake, need to select one per FU

Tomasulo's Algorithm for OoO Execution

- OoO with register renaming invented by Robert Tomasulo
  - Used in IBM 360/91 Floating Point Units
  - **Read:** Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of R&D, Jan. 1967.

- OoO variants are used in most high-performance processors
  - Initially in Intel Pentium Pro, AMD K5
  - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15

Recall: Register Renaming

- WAR and WAW dependencies are not true dependencies
  - WHY? The same register refers to values that have nothing to do with each other
  - **They exist because not enough register ID's (i.e. names) in the ISA**

- How register renaming eliminates WAR and WAW dependencies?
  - Require: a large number of (physical) registers except for the architecture registers

Register Renaming Example

- "Architected" vs "Physical" registers – level of indirection
  - Arch: **`r1,r2,r3`**
  - Physical: **`p1,p2,p3,p4,p5,p6,p7`**

Original insns

Time

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

Register Renaming Example

- "Architected" vs "Physical" registers – level of indirection
  - Arch: `r1,r2,r3`
  - Physical: `p1,p2,p3,p4,p5,p6,p7`

Original insns

Time

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

MapTable

Time

r1	r2	r3
p1	p2	p3

FreeList

p4,p5,p6,p7

Original insns

```
add r2,r3➜r1
```

Renamed insns

```
add p2,p3➜p4
```

Register Renaming Example

- "Architected" vs "Physical" registers – level of indirection
  - Arch: `r1,r2,r3`
  - Physical: `p1,p2,p3,p4,p5,p6,p7`

Original insns

Time

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

MapTable

Time

r1	r2	r3
p1	p2	p3
p4	p2	p3

FreeList

p4,p5,p6,p7
p5,p6,p7

Original insns

```
add r2,r3➜r1
sub r2,r1➜r3
```

Renamed insns

```
add p2,p3➜p4
sub p2,p4➜p5
```

Register Renaming Example

- "Architected" vs "Physical" registers – level of indirection
  - Arch: `r1,r2,r3`
  - Physical: `p1,p2,p3,p4,p5,p6,p7`

Original insns

Time

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

MapTable

Time

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5

FreeList

p4,p5,p6,p7
p5,p6,p7
p6,p7

Original insns

```
add r2,r3➜r1
sub r2,r1➜r3
mul r2,r3➜r3
```

Renamed insns

```
add p2,p3➜p4
sub p2,p4➜p5
mul p2,p5➜p6
```

Register Renaming Example

- "Architected" vs "Physical" registers – level of indirection
  - Arch: `r1,r2,r3`
  - Physical: `p1,p2,p3,p4,p5,p6,p7`

Original insns

Time

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

+ Removes **false** dependences
+ Leaves **true** dependences intact!

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
```

Tomasulo's Algorithm for OoO Execution

Key Technologies

- Each compute unit maintains a reservation station (physical registers)
  - buffer the insts who are not ready
- Register renaming to remove WAW and WAR dependencies
  - Rename Register ID to RS entry ID
- Out of Order dispatching

Tomasulo's Algorithm

Reservation Station

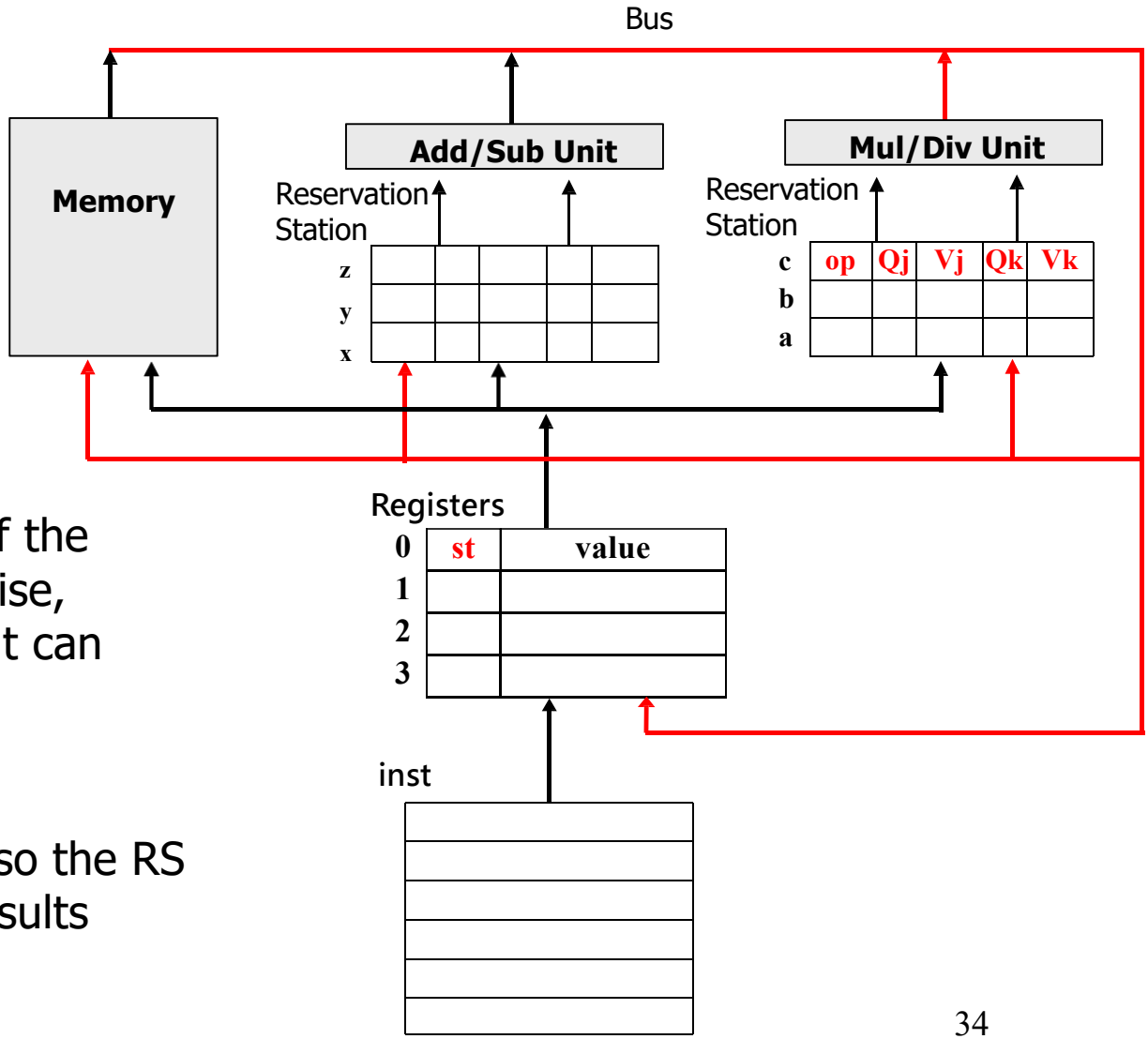
- Op: op code of the inst
- Vj, Vk: values of operants
- Qj, Qk: RS entry ID that will provide the value (0 means the value is ready)

Registers (add a new field):

- st: empty means the value of the register can be used, otherwise, indicates the RS entry ID that can provide the value

Bus

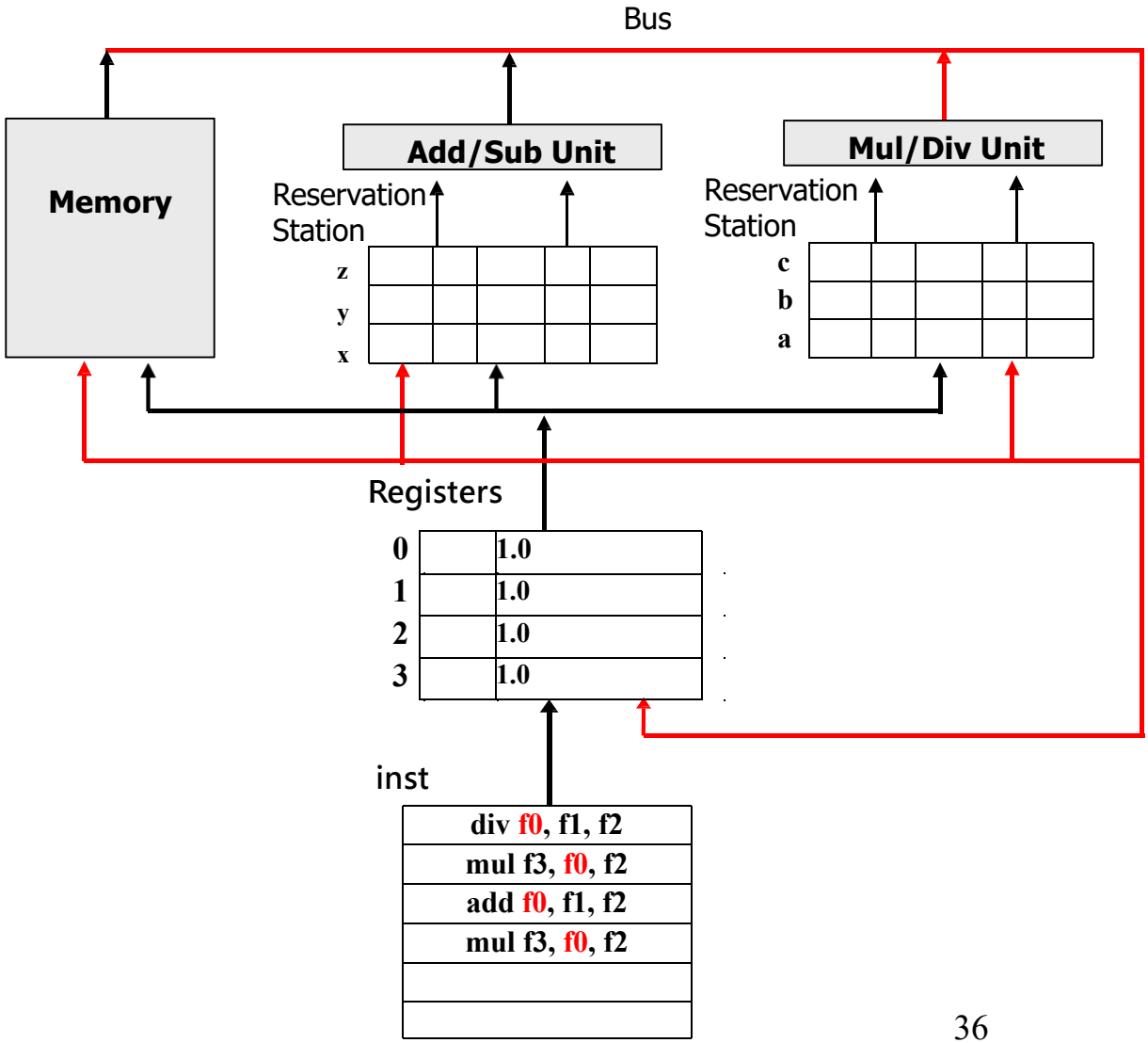
- broadcast the results, and also the RS entry ID that provides the results



Tomasulo's Algorithm

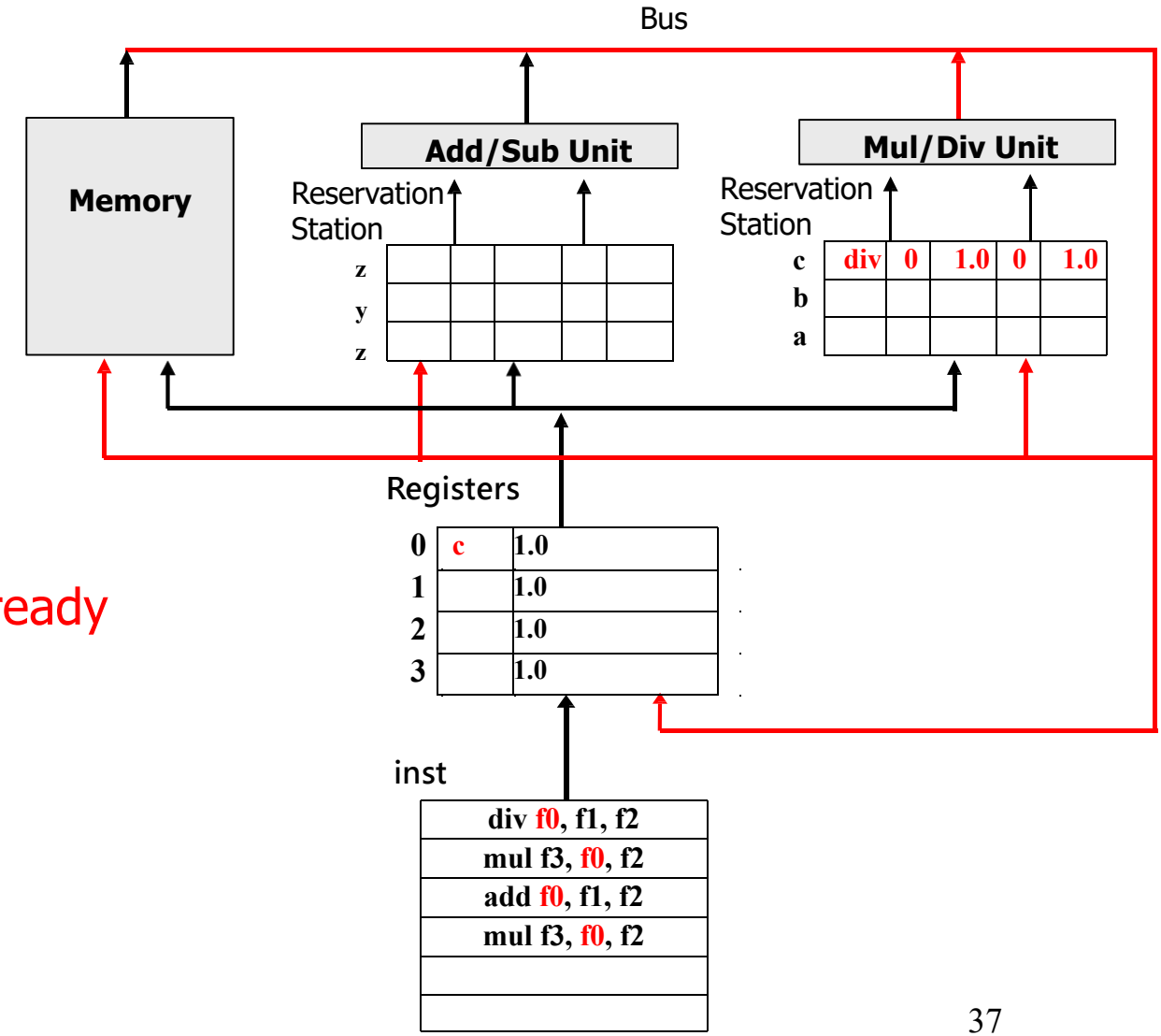
- If reservation station available before renaming
  - Instruction + renamed operands (source value/tag) inserted into the reservation station
  - Only rename if reservation station is available
- Else stall
- While in reservation station, each instruction:
  - Watches common data bus (CDB) for tag of its sources
  - When tag seen, grab value for the source and keep it in the reservation station
  - When both operands available, instruction ready to be dispatched
- Dispatch instruction to the Functional Unit when instruction is ready
- After instruction finishes in the Functional Unit
  - Arbitrate for CDB
  - Put tagged value onto CDB (tag broadcast)
  - Register file is connected to the CDB
    - Register contains a tag indicating the latest writer to the register
    - If the tag in the register file matches the broadcast tag, write broadcast value into register (and set valid bit)
  - Reclaim rename tag
    - no valid copy of tag in system!

Tomasulo's Algorithm Example

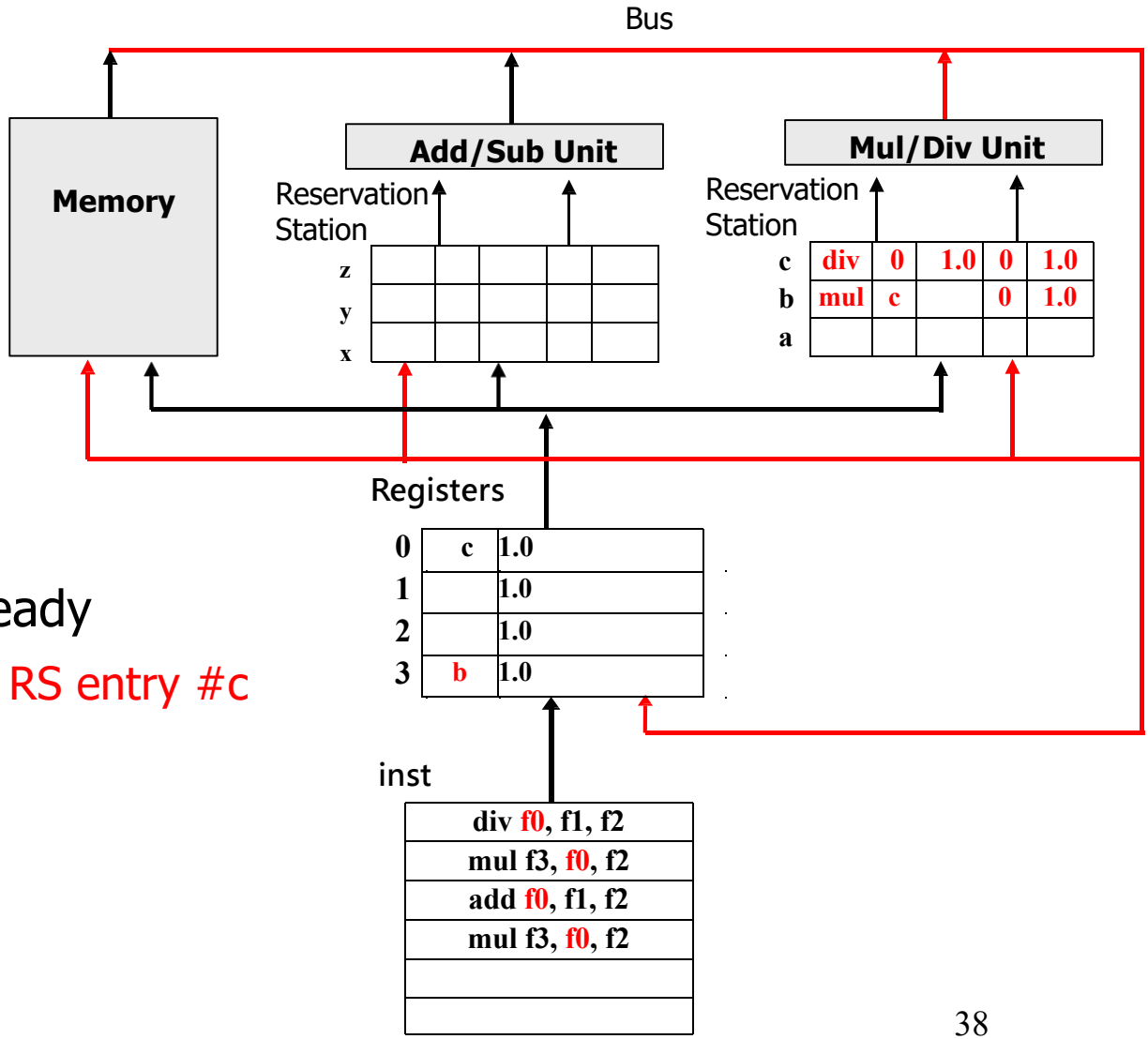


Tomasulo's Algorithm Example

div issue, f1 and f2 are ready



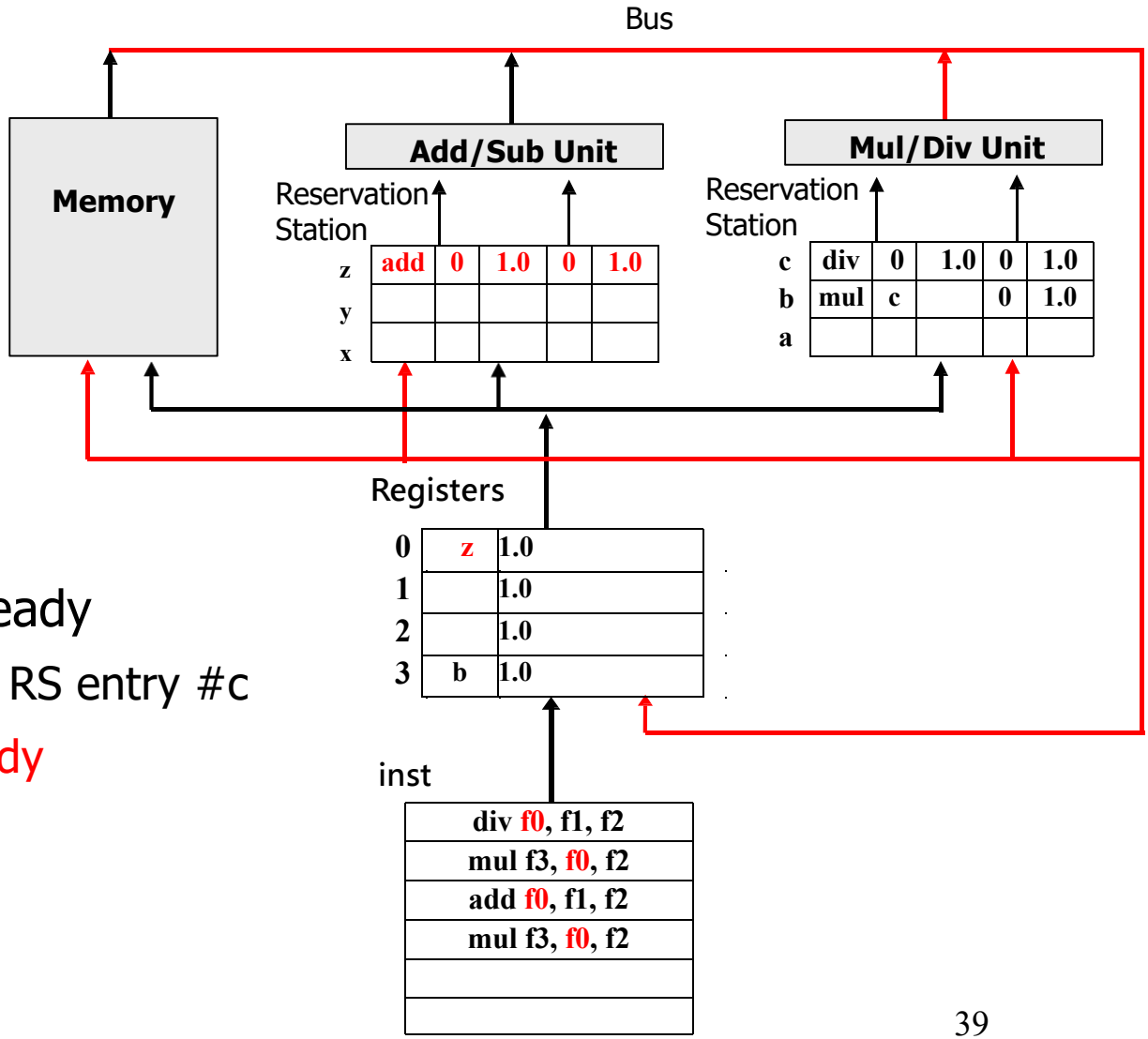
Tomasulo's Algorithm Example



div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for RS entry #c

Tomasulo's Algorithm Example

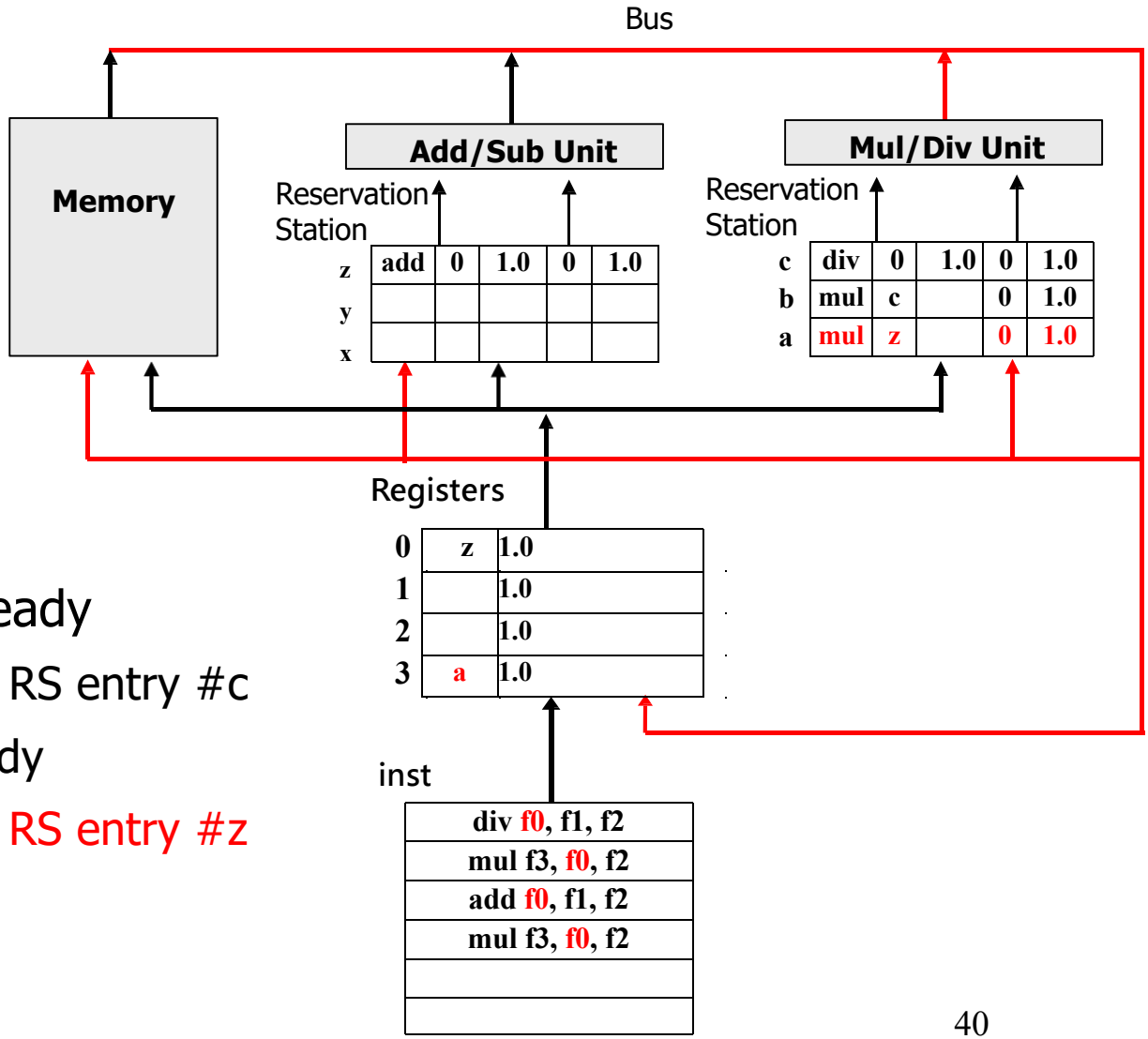


div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for RS entry #c

add issue, f1 and f2 are ready

Tomasulo's Algorithm Example



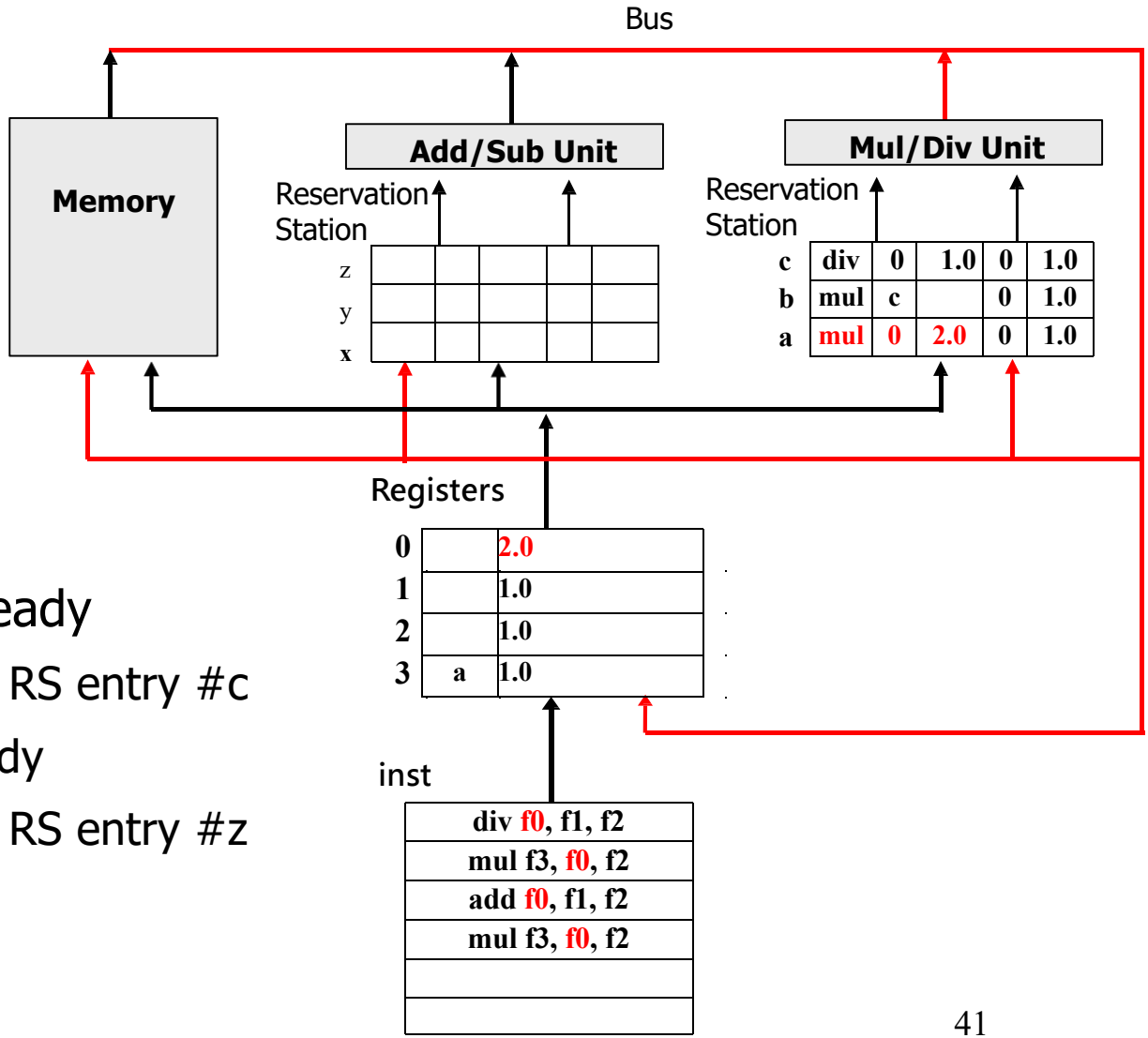
div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for RS entry #c

add issue, f1 and f2 are ready

mul2 issue, f0 is waiting for RS entry #z

Tomasulo's Algorithm Example



div issue, f1 and f2 are ready

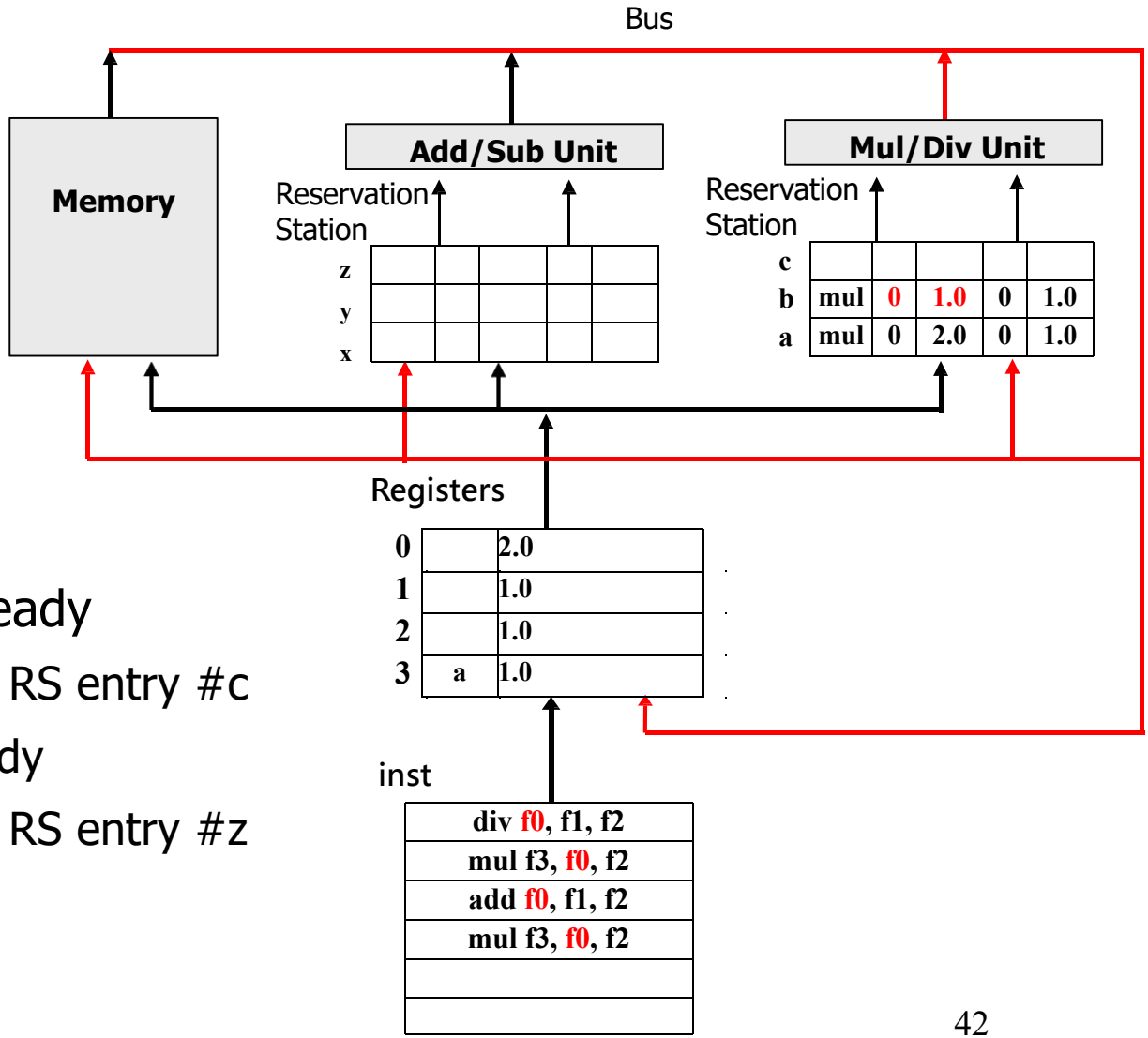
mul1 issue, f0 is waiting for RS entry #c

add issue, f1 and f2 are ready

mul2 issue, f0 is waiting for RS entry #z

add write back

Tomasulo's Algorithm Example



div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for RS entry #c

add issue, f1 and f2 are ready

mul2 issue, f0 is waiting for RS entry #z

add write back

div write back

Exception

Exceptions in Pipeline

- I/O request：external interrupt
- Inst Exception：user request interrupt
  - sys call、breakpoint、trace and debug
- Compute Units
  - overflow、float exception
- Storage Units
  - unaligned memory address、invalid access to system space、TLB miss、page fault
- Reserved Inst Error：unknown inst
- Hardware errors
- etc

Precise Exception

- **Precise Exception**: when exception happens, the insts before exception should be executed, and the insts after exception are not executed
- **Reason for imprecise exception:** out of order, later insts finish earlier and modify the system states (registers or memory)
  - Tomasulo Algorithm cannot ensure precise exception

DIVF f0,f2,f4

ADDF f10,f10,f8

SUBF f12,f12,f14

addf and subf may finish earlier than divf, if exception happens for divf after addf is executed, it is imprecise

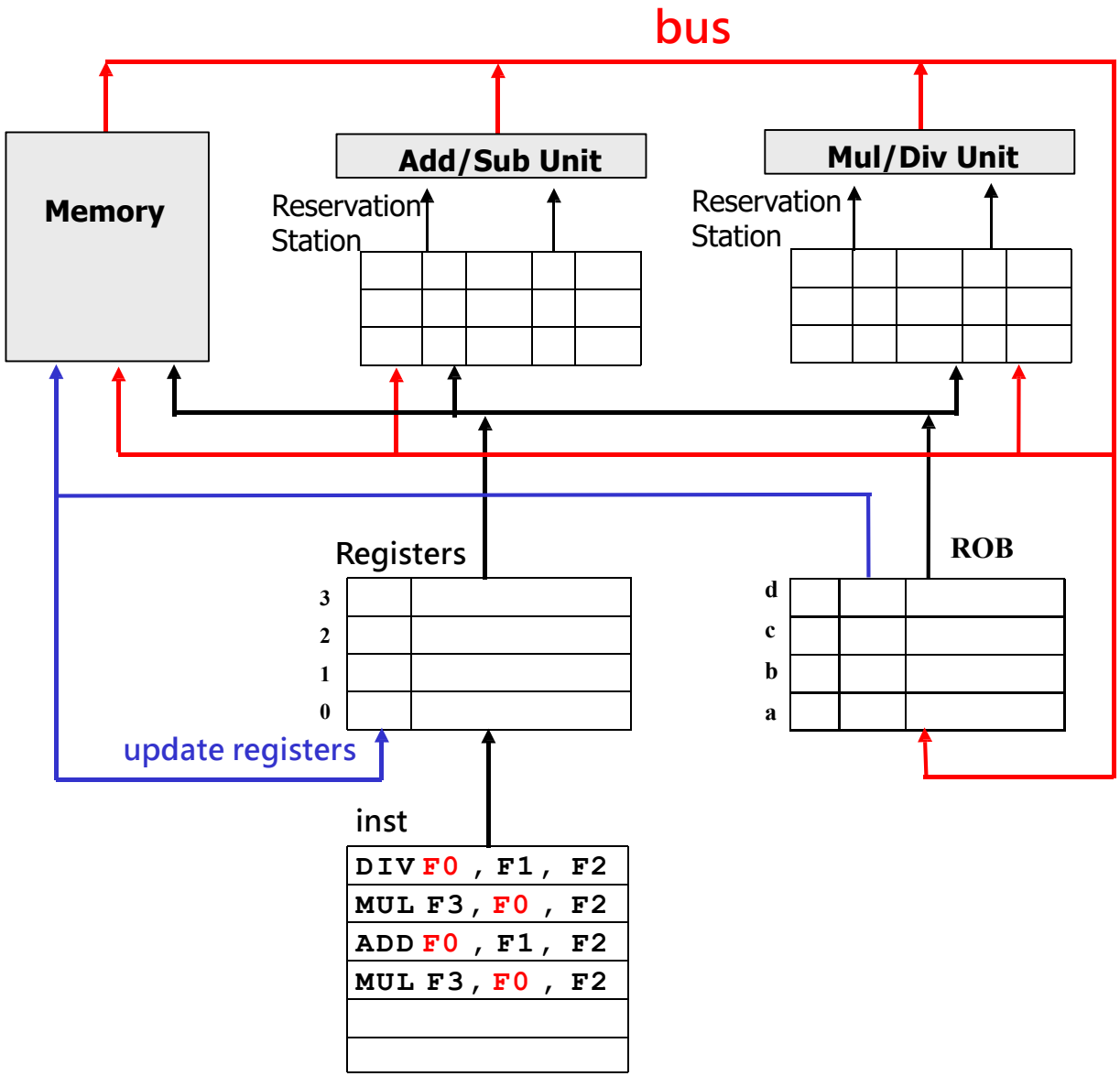
Precise Exception for OoO Pipepline

- **Idea:** use buffers to store execution results temporarily, commit a result when all the previous insts' results are commited
  - **Reorder Buffer (ROB):** buffer the execution results
  - when an inst writes back, put the results in ROB
  - add a **Commit** phase, write the results in ROB to register/memory
  - An inst is only committed when all the previous insts are committed
  - **In-Order Commit**:**Out-of-Order Execution**, **In-Order Complete**

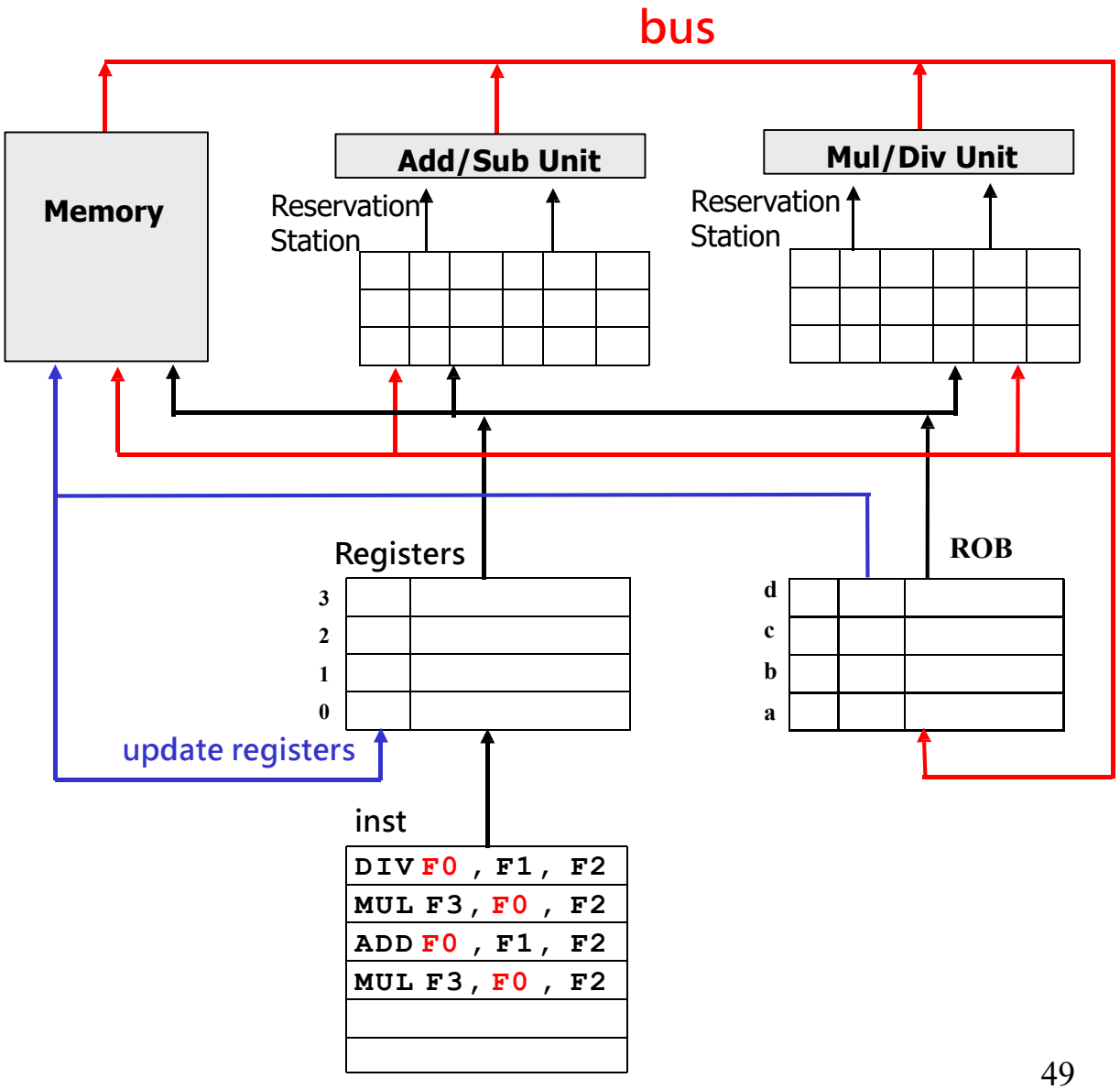
Reorder Buffer（ROB）

- **Content:** target address（register ID）、value、op
- Write the results to ROB in the WriteBack phase, later insts can read from operants from ROB
- **Register ID is renamed to ROB ID (not the RS ID)**
- Write the results to register/memory when commit
- As long as an inst is not committed, it will not modify the registers/memory, easy to cancel the inst before it is committed（due to exceptions of ealier insts

Pipeline with Reorder Buffer

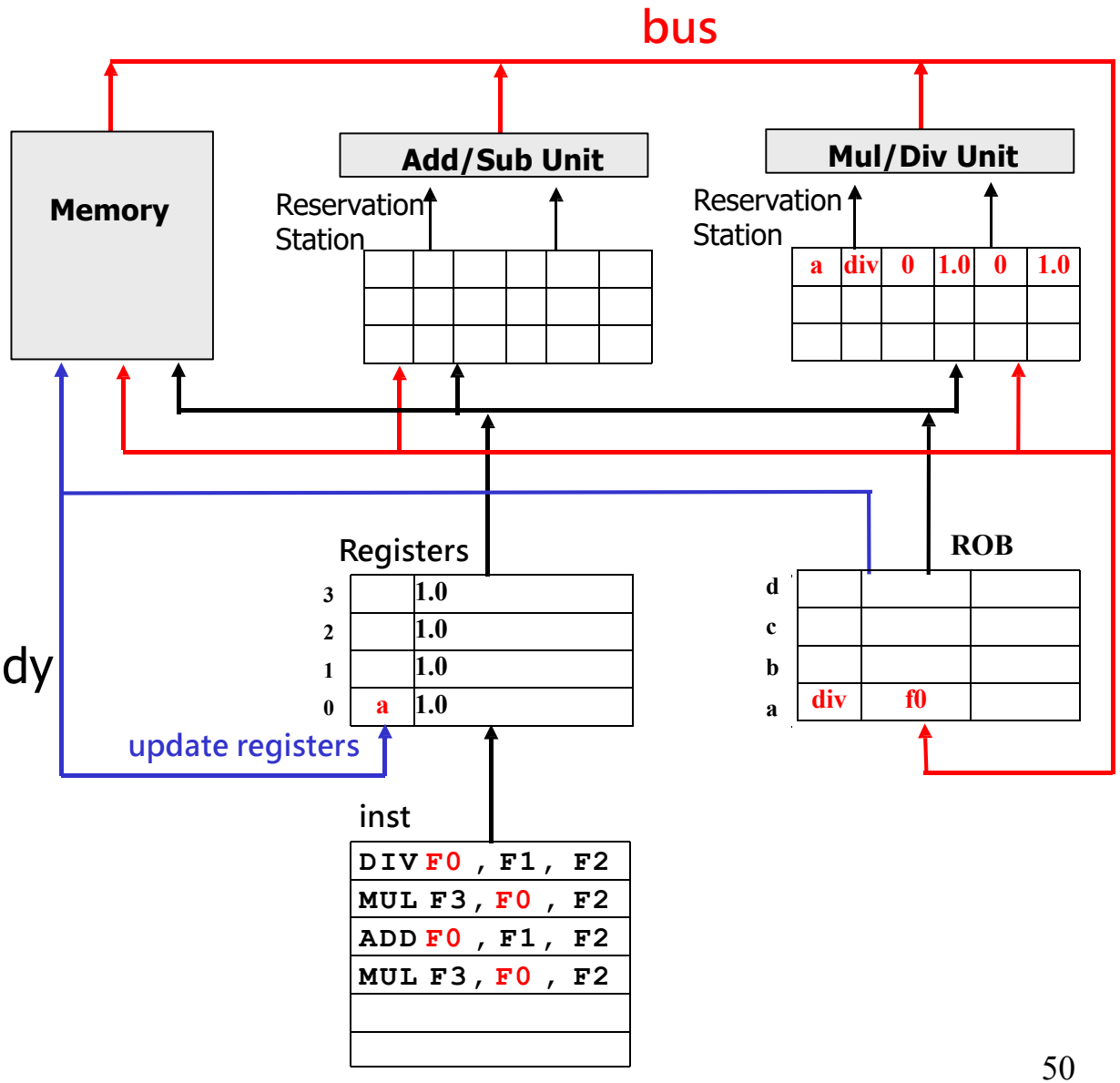


```
DIV  F0,F1, F2
MUL  F3,F0, F2
ADD  F0,F1, F2
MUL  F3,F0, F2
```

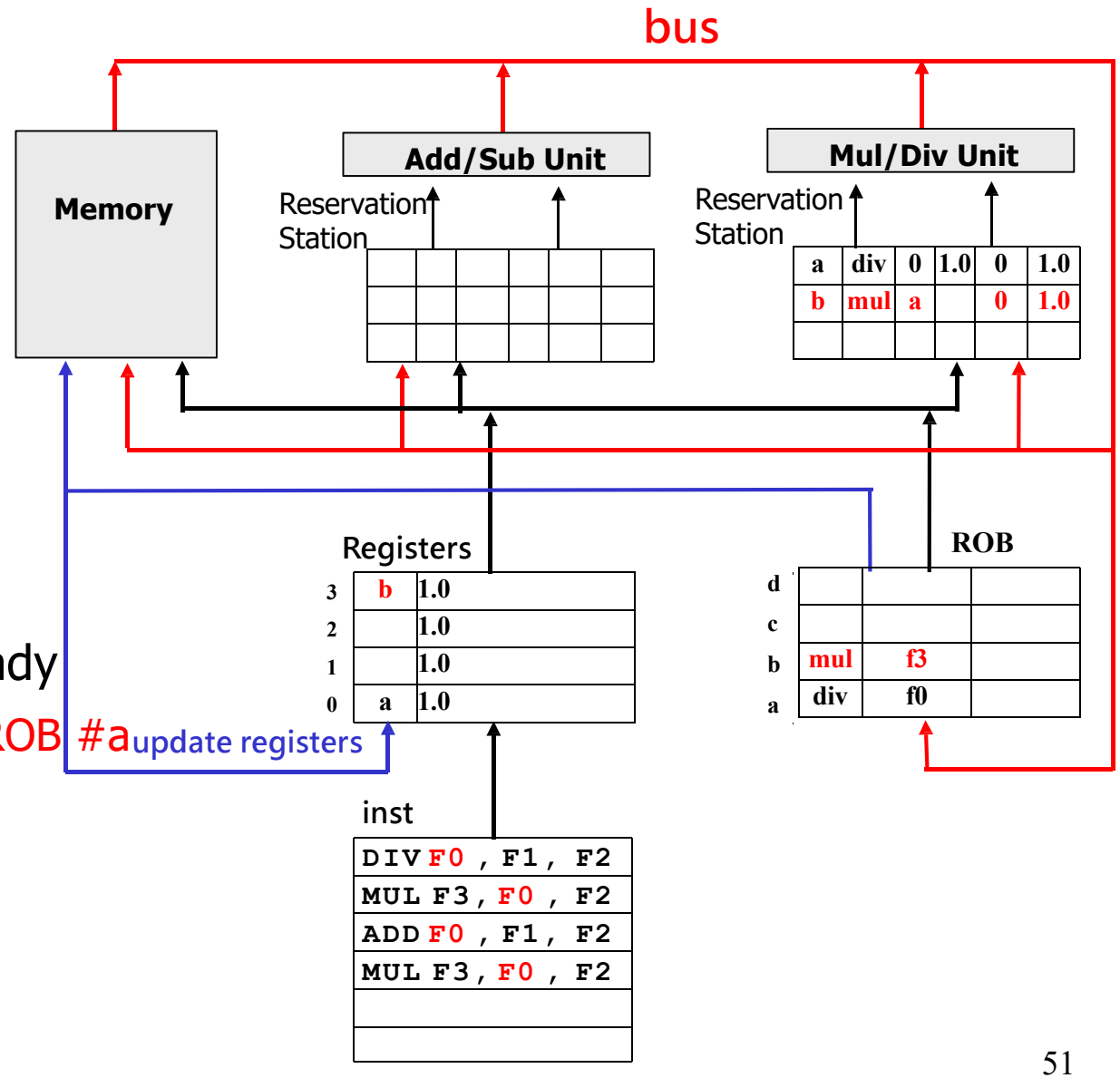


```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```

div issue, f1 and f2 are ready



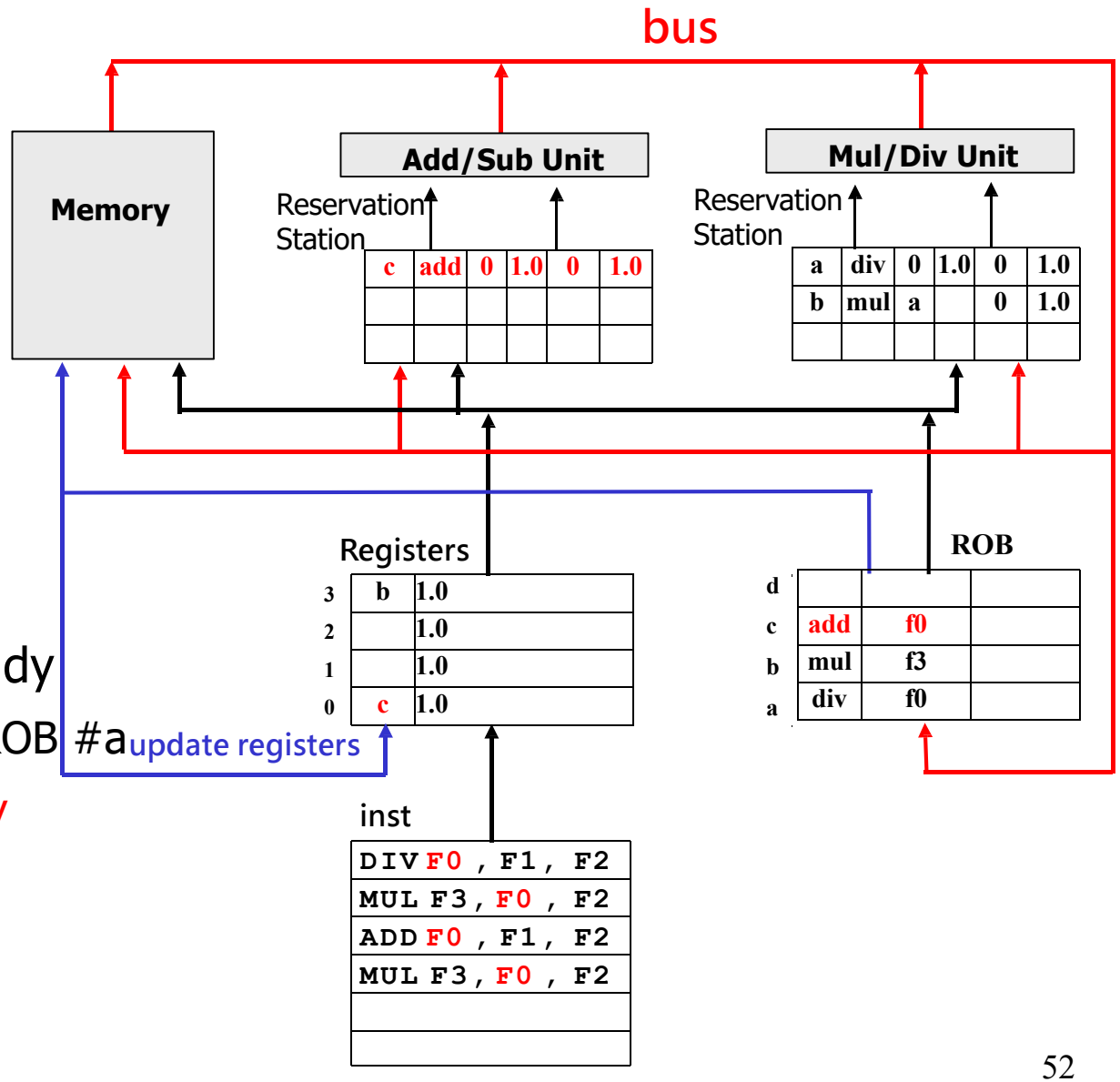
```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```



div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a update registers

```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```

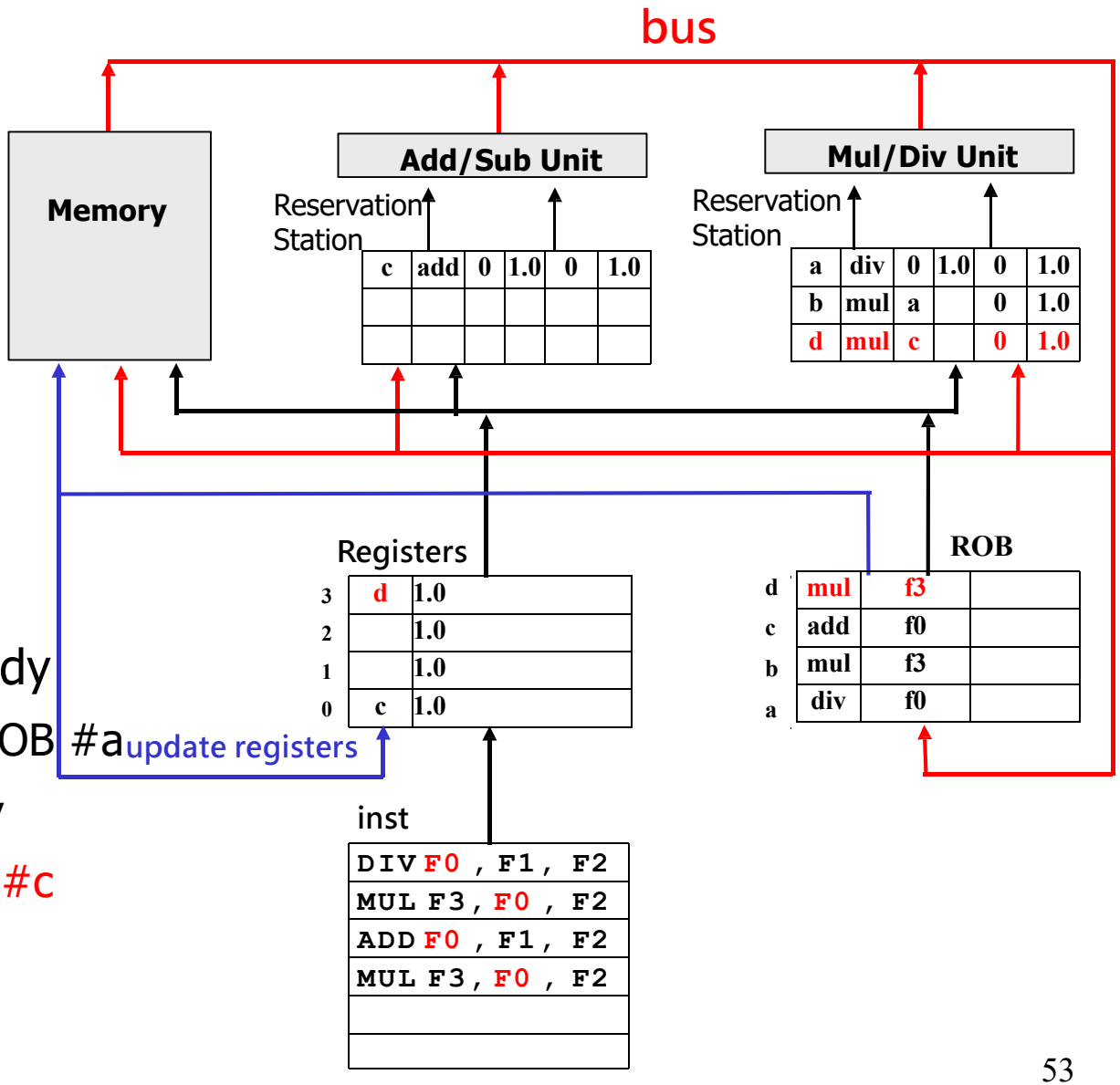


div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a

add issue, f1 and f2 are ready

```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```



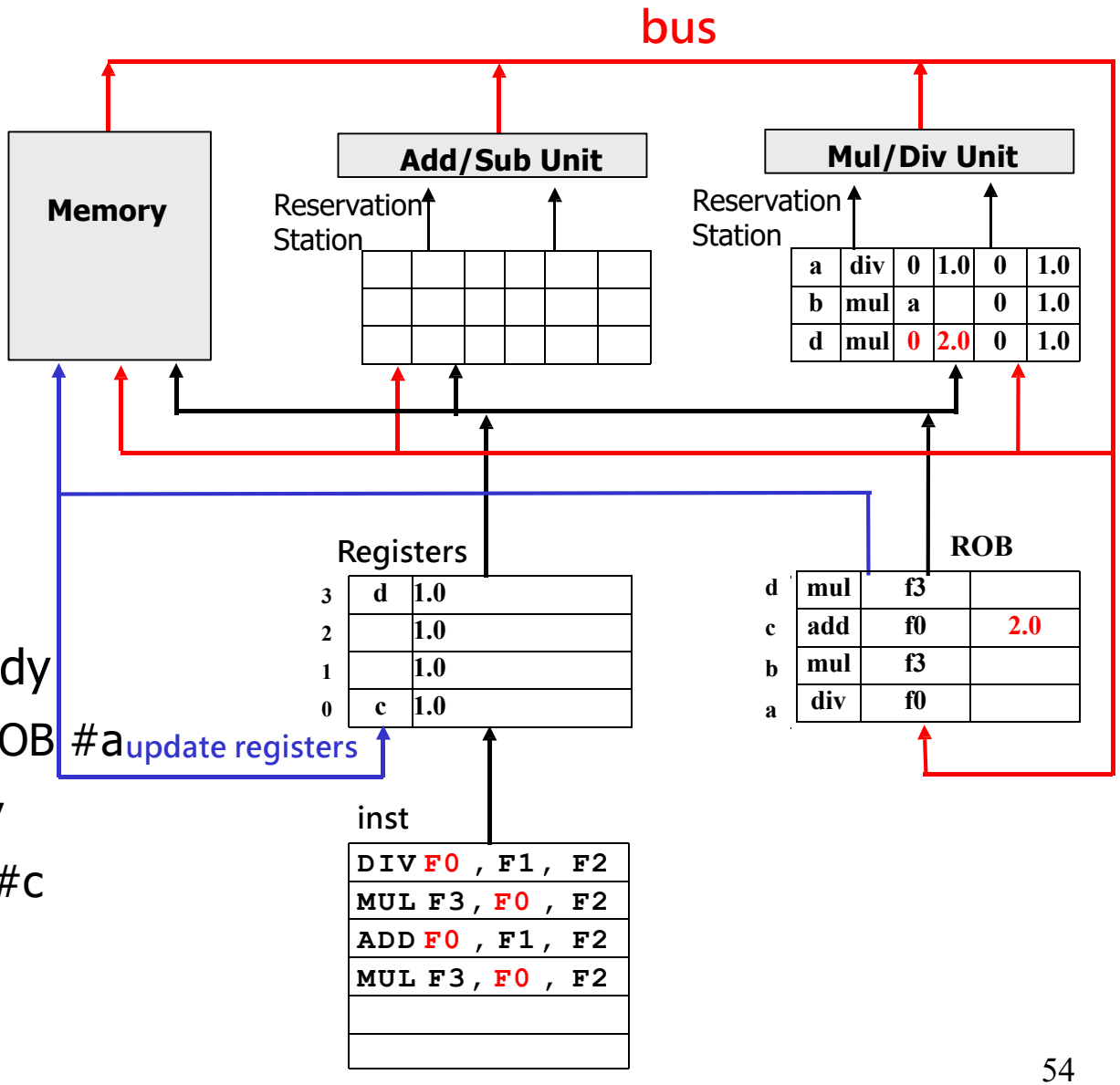
div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a

add issue, f1 and f2 are ready

mul2 issue, f0 is waiting ROB #c

```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```



div issue, f1 and f2 are ready

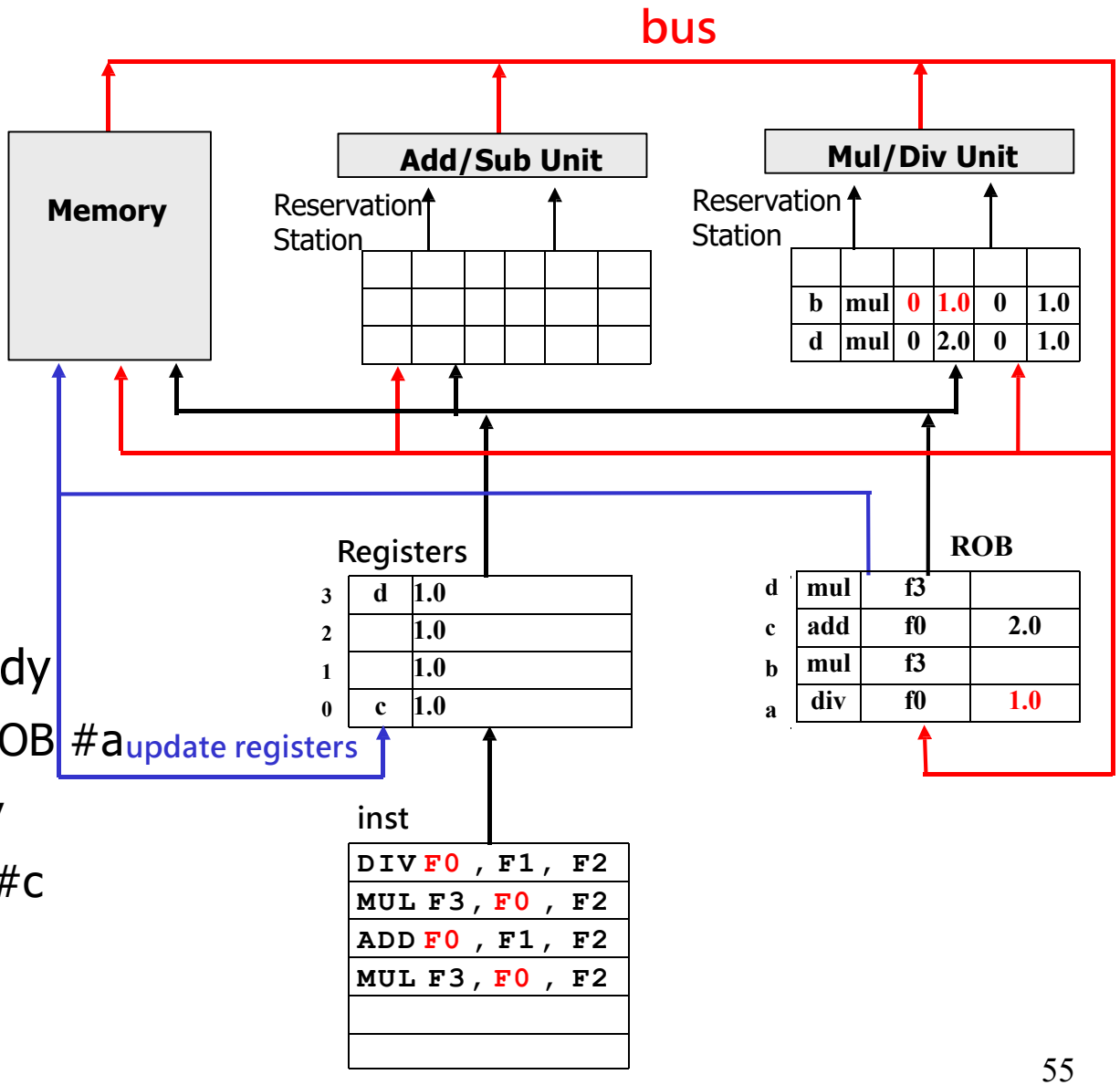
mul1 issue, f0 is waiting for ROB #a

add issue, f1 and f2 are ready

mul2 issue, f0 is wating ROB #c

add write back

```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```



div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a

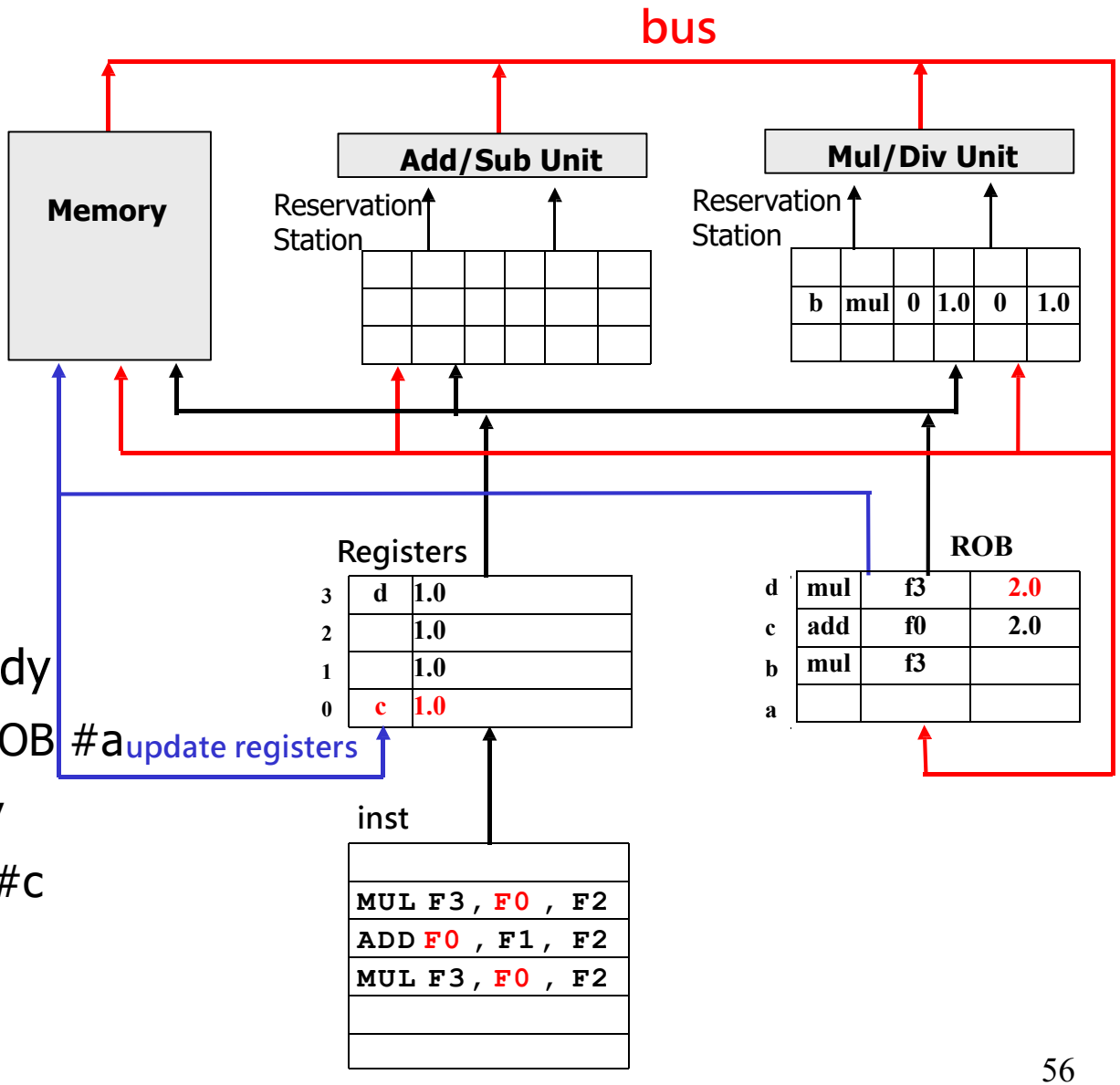
add issue, f1 and f2 are ready

mul2 issue, f0 is wating ROB #c

add write back

div write back

```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```



div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a

add issue, f1 and f2 are ready

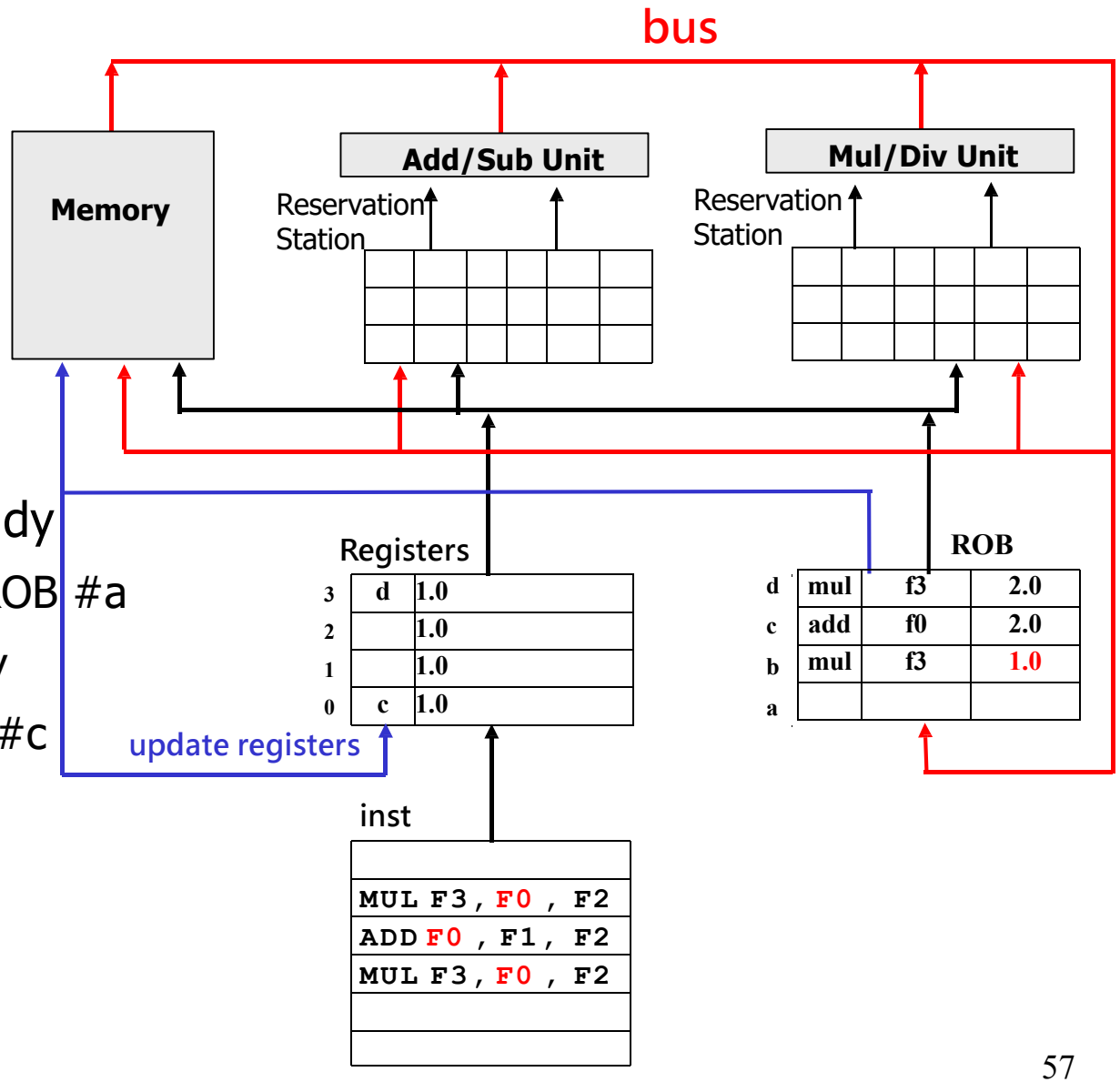
mul2 issue, f0 is wating ROB #c

add write back

div write back

div commit, mul2 write back

```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```



div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a

add issue, f1 and f2 are ready

mul2 issue, f0 is wating ROB #c

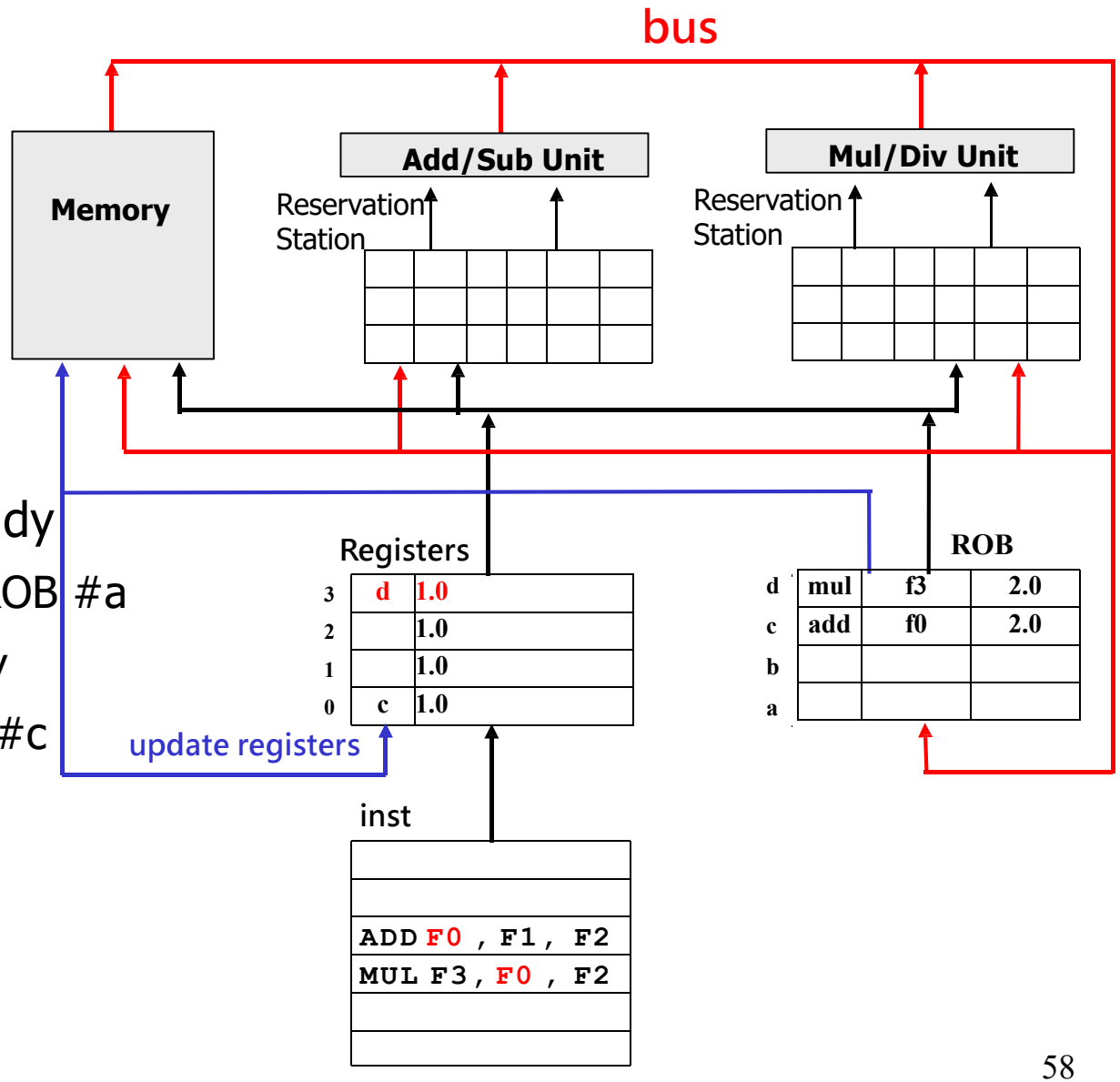
add write back

div write back

div commit, mul2 write back

mul write back

```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```



div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a

add issue, f1 and f2 are ready

mul2 issue, f0 is wating ROB #c

add write back

div write back

div commit, mul2 write back

mul1 write back

mul1 commit

```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```

div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a

add issue, f1 and f2 are ready

mul2 issue, f0 is wating ROB #c

add write back

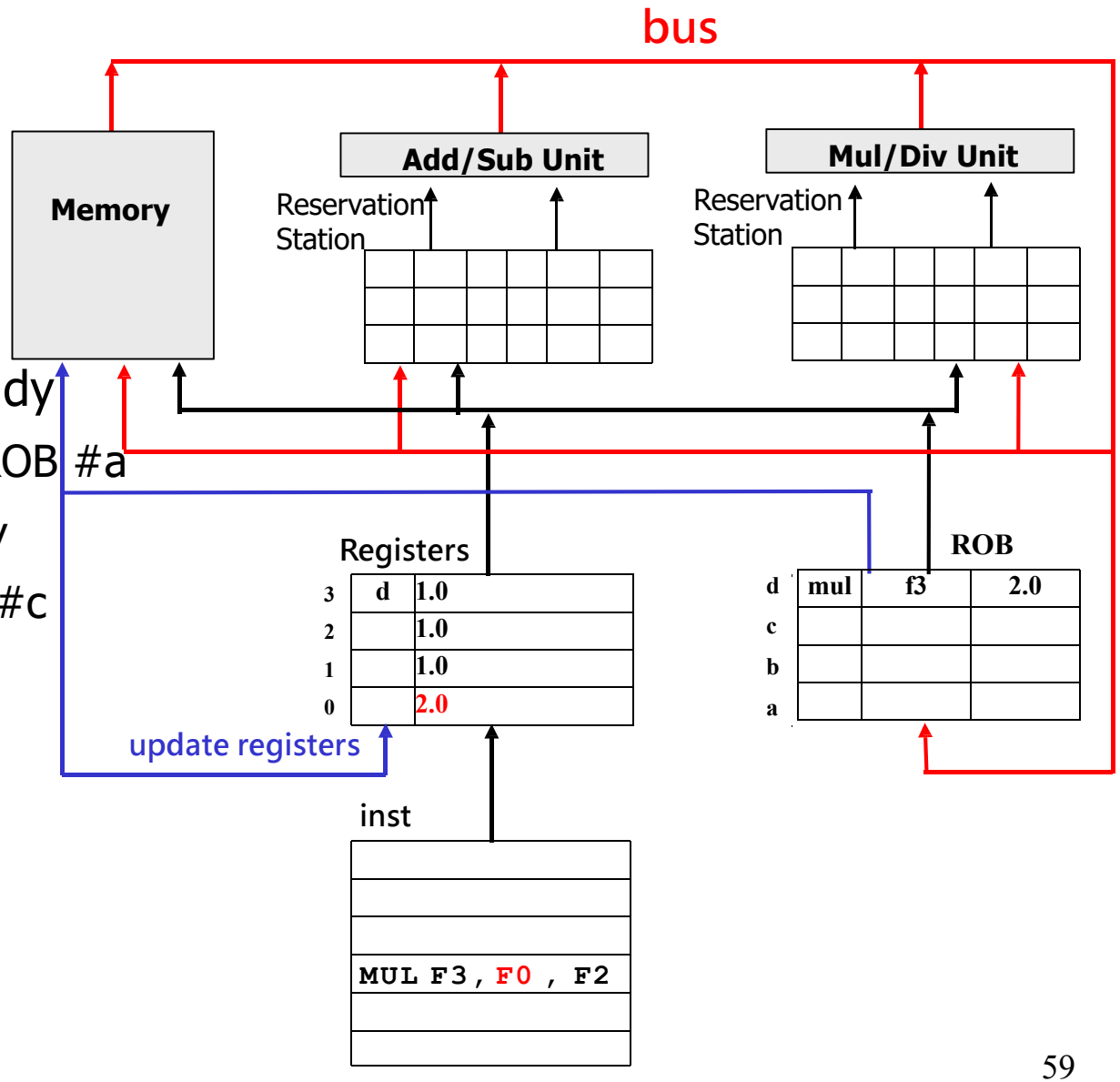
div write back

div commit, mul2 write back

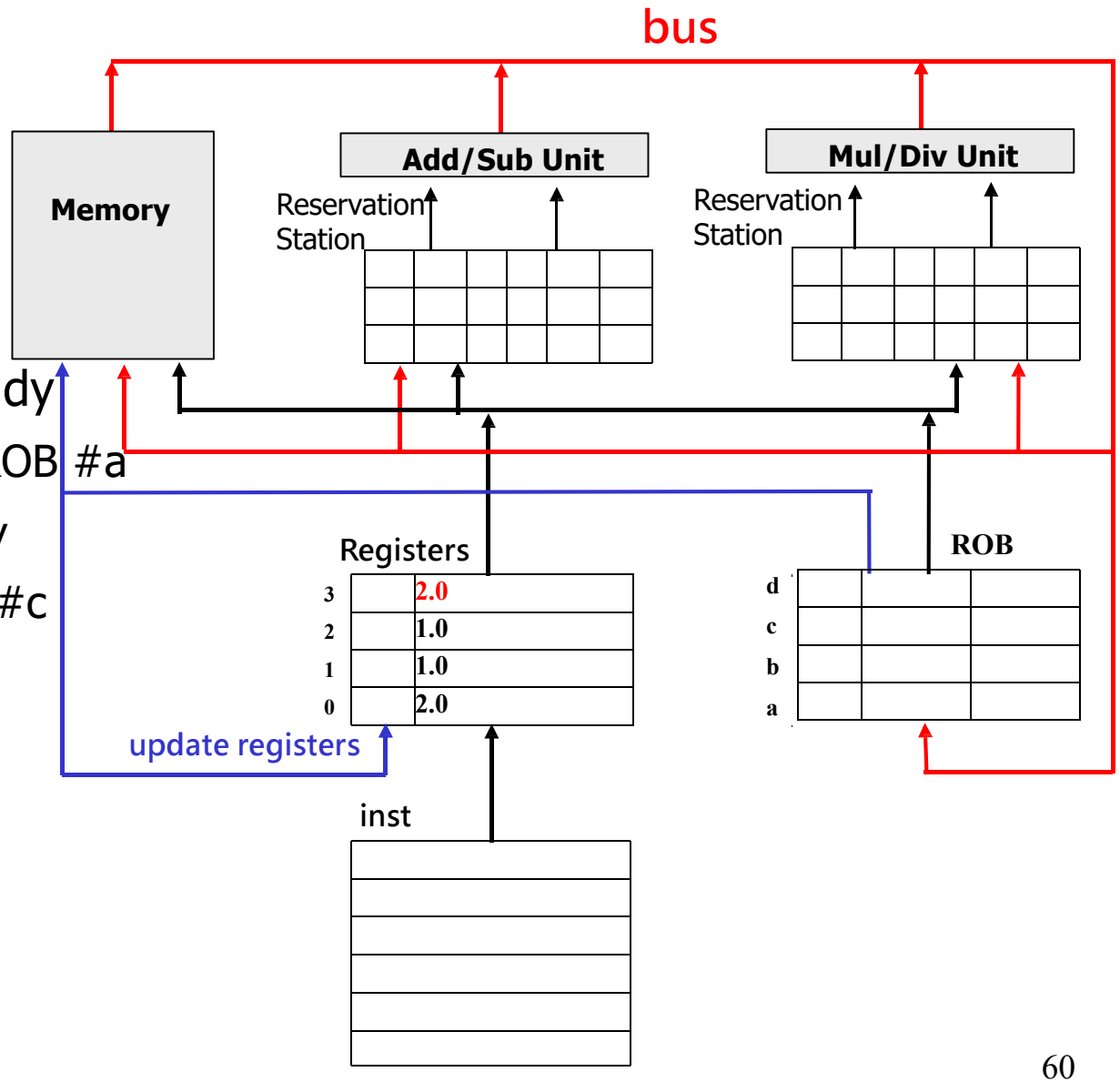
mul1 write back

mul1 commit

add commit



```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```



div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a

add issue, f1 and f2 are ready

mul2 issue, f0 is wating ROB #c

add write back

div write back

div commit, mul2 write back

mul1 write back

mul1 commit

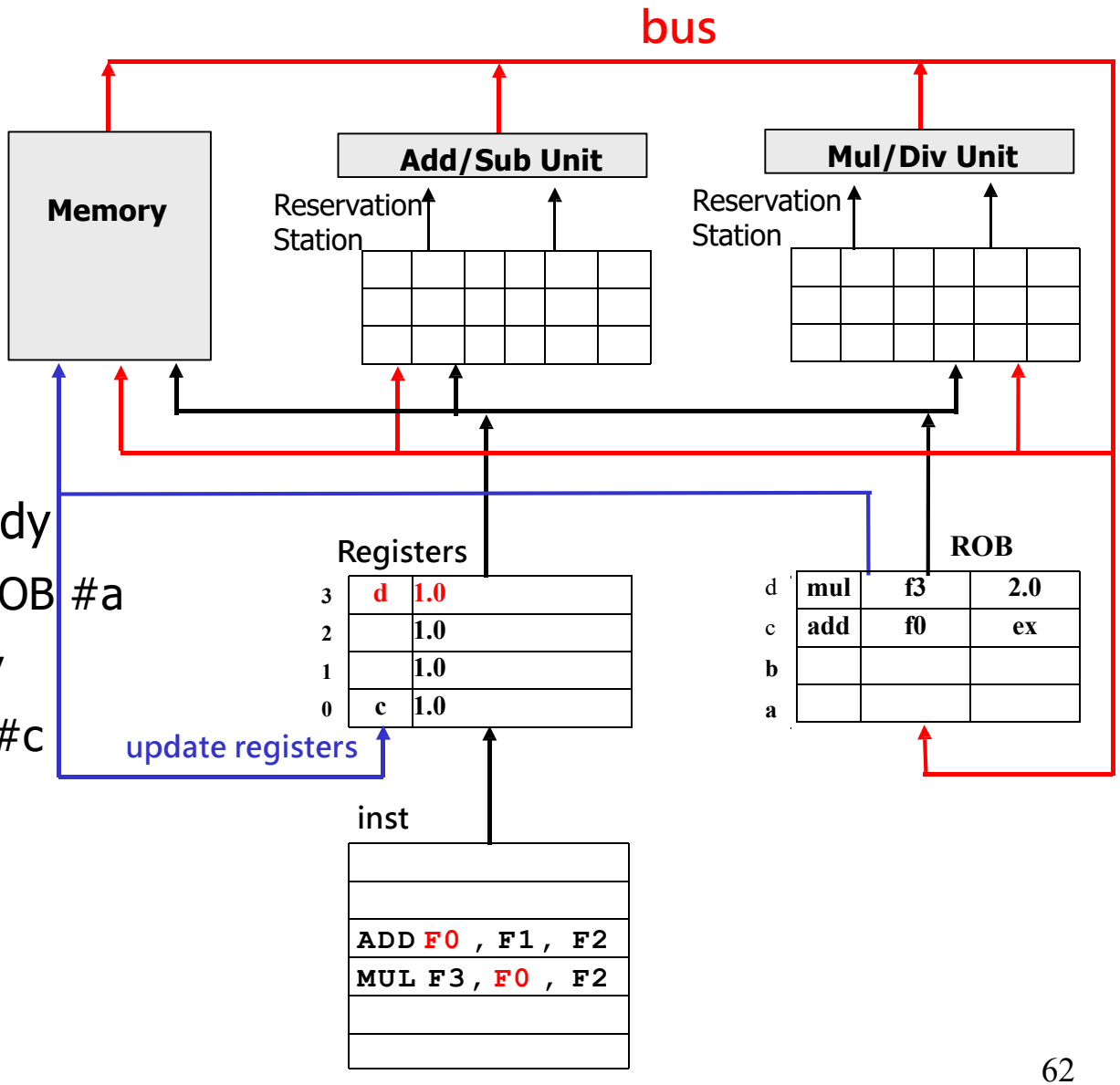
add commit

mult2 commit

ROB with Exception

suppose ADD has execption

```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```



div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a

add issue, f1 and f2 are ready

mul2 issue, f0 is wating ROB #c

add write back

div write back

div commit, mul2 write back

mul1 write back

mul1 commit

suppose ADD has execption

```
DIV  F0, F1, F2
MUL  F3, F0, F2
ADD  F0, F1, F2
MUL  F3, F0, F2
```

div issue, f1 and f2 are ready

mul1 issue, f0 is waiting for ROB #a

add issue, f1 and f2 are ready

mul2 issue, f0 is wating ROB #c

add write back

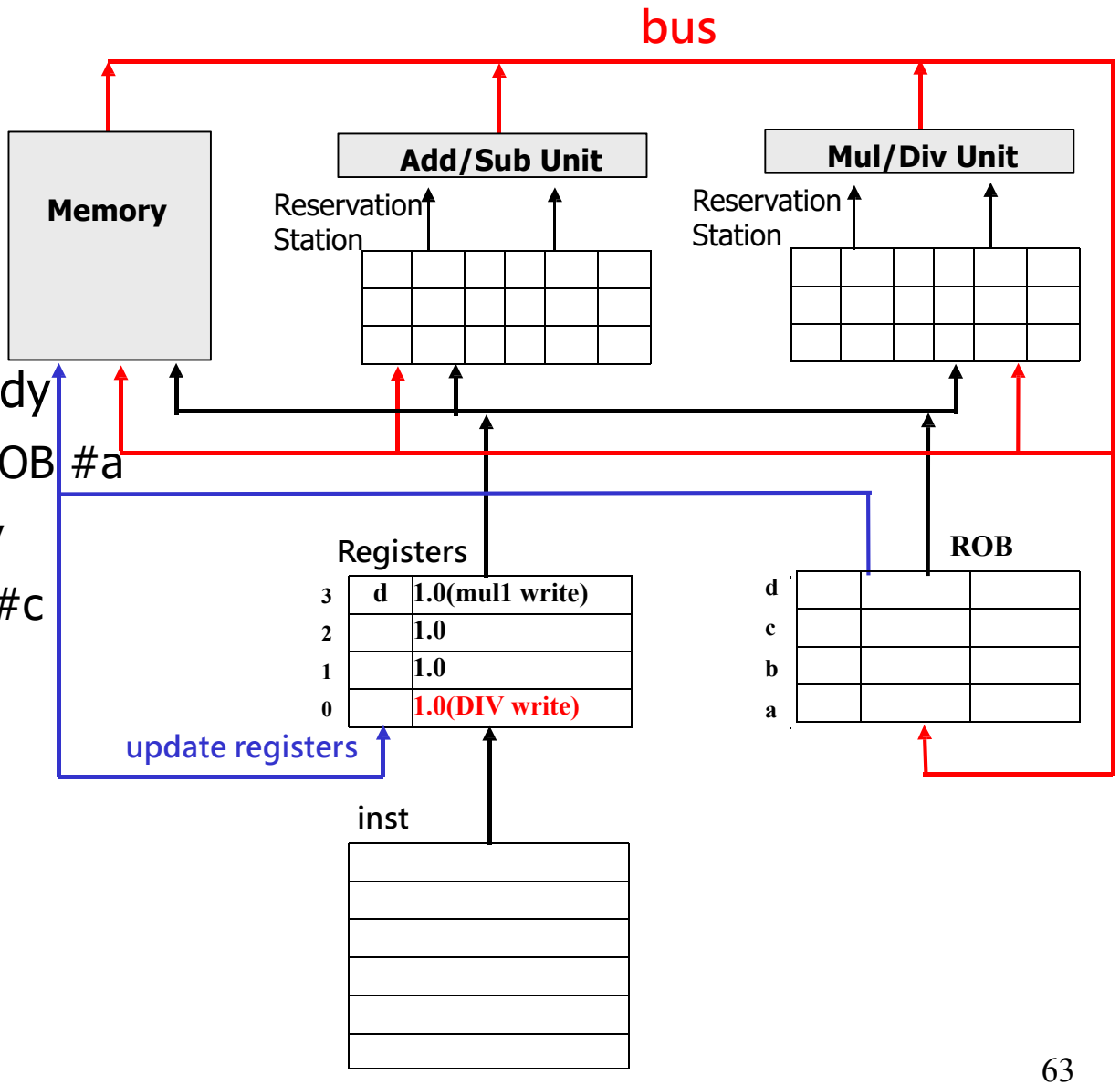
div write back

div commit, mul2 write back

mul1 write back

mul1 commit

add commit



OoO Pipeline with ROB

- **Issue**：send the inst to RS (if both RS and ROB are available)，use a ROB entry to store the result；read the operants (check the ready status)
- **Execute**：if all the operants are ready，execute the inst, otherwise watch common data bus for ROB tag of its sources; when tag seen, grab value for the source and keep it in the reservation station
- **Write back**：send the results to data bus，release RS entry；ROB updates the results accordingly
- **Commit**：if the earlies inst writes back and there is no exception，write the results in ROB to register/memory (i.e., commit)，release the ROB entry; if the earlier inst has exception, empty the insts list and ROB etc

Handling Memory Operations

Recall: Types of Dependencies

- RAW (Read After Write) = "true dependence"

  mul r0 * r1 ➔ **r2**

  …

  add **r2** + r3 ➔ r4

- WAW (Write After Write) = "output dependence"

  mul r0 * r1➔ **r2**

  …

  add r1 + r3 ➔ **r2**

- WAR (Write After Read) = "anti-dependence"

  mul r0 * **r1** ➔ r2

  …

  add r3 + r4 ➔ **r1**

- WAW & WAR are "false", Can be **totally eliminated** by "renaming"

Also Have Dependencies via Memory

- **If value in "r2" and "r3" is the same…**
- RAW (Read After Write) – True dependency

  st r1 ➜ **[r2]**

  …

  ld **[r3]** ➜ r4
- WAW (Write After Write)

  st r1 ➜ **[r2]**

  …

  st r4 ➜ **[r3]**
- WAR (Write After Read)

  ld **[r2]** ➜ r1

  …

  st r4 ➜ **[r3]**

WAR/WAW are "false dependencies"

- But can't rename memory in same way as registers
  - **Why? Addresses are not known at rename**
- Need to use other tricks

Let's Start with Just Stores

- Stores: Write data cache, not registers
  - Can we rename memory?
    - ➢ No (at least not easily)
  - Cache writes unrecoverable
- Solution: write stores into cache only when certain
  - When are we certain? At "commit"

Handling Stores

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 ➜ p3	F	Di	I	RR	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	RR	X	W	C		
st p5 ➜ [p3+4]		F	Di				I	RR	X	M	W	C	
st p4 ➜ [p6+8]		F	Di	I?									

- Can "st p4 ➜ [p6+8]" issue in cycle 3?
  - Its register inputs are ready…

Problem #1: Out-of-Order Stores

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 ➜ p3	F	Di	I	RR	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	RR	X	W	C		
st p5 ➜ [p3+4]		F	Di				I	RR	X	**M**	W	C	
st p4 ➜ [p6+8]		F	Di	**I?**	**RR**	**X**	**M**	**W**				C	

- Can "st p4 ➜ [p6+8]" write the cache in cycle 6?
  - "st p5 ➜ [p3+4]" has not yet executed
- What if p3+4 == p6+8?
  - The two stores write the same address! **WAW dependency!**
  - Not known until their "X" stages (cycle 5 & 8)
- Unappealing solution: all stores execute in-order
- We can do better…

Problem #2: Speculative Stores

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 ➜ p3	F	Di	I	RR	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	RR	~~X~~	W	C		
st p5 ➜ [p3+4]		F	Di				I	RR	X	M	W	C	
st p4 ➜ [p6+8]		F	Di	**I?**	**RR**	**X**	**M**	**W**				**C**	

- Can "st p4 ➜ [p6+8]" write the cache in cycle 6?
  - Store is still "speculative" at this point
- What if "jump-not-zero" is mis-predicted?
  - Not known until its "X" stage (cycle 8)
- How does it "undo" the store once it hits the cache?
  - Answer: it can't; stores write the cache only at **commit**
  - Guaranteed to be non-speculative at that point

Store Queue (SQ)

- Solves two problems
  - Allows for recovery of speculative stores
  - Allows out-of-order stores
- Store Queue (SQ)
  - **At dispatch, each store is given a slot in the Store Queue**
  - First-in-first-out (FIFO) queue
  - Each entry contains: “address”, “value”, and “bday”
- Operation:
  - Dispatch (in-order): allocate entry in SQ (stall if full)
  - Execute (out-of-order): write store value into store queue
  - Commit (in-order): read value from SQ and write into data cache
  - Branch recovery: remove entries from the store queue
- Also solves problems with loads

Store Queue Operation

	0	1	2	3	4	5	6	7	8	9	10	11	12
fdiv p1 / p2 ➜ p9	F	Di	I	RR	X_1	X_2	X_3	X_4	X_5	X_6	W	C	
st p4 ➜ [p5+4]	F	Di	I	RR	X	SQ						C	
st p3 ➜ [p6+8]		F	Di	I	RR	X	SQ						C

- Stores write to SQ, not M
  - similar to register renaming, where we allocated a new physical register for each insn
- What if the fdiv triggers a /0 exception?

Memory Forwarding

	0	1	2	3	4	5	6	7	8	9	10	11	12
fdiv p1 / p2 ➜ p9	F	Di	I	RR	X_1	X_2	X_3	X_4	X_5	X_6	W	C	
st p4 ➜ [p5+4]	F	Di	I	RR	X	SQ						C	
st p3 ➜ [p6+8]		F	Di	I	RR	X	SQ						C
ld [p7] ➜ p8		F	Di	I?	RR	X	M_1	M_2	W				C

- Can "ld [p7] ➜ p8" issue and begin execution?
  - Why or why not?

Memory Forwarding

	0	1	2	3	4	5	6	7	8	9	10	11	12
fdiv p1 / p2 ➜ p9	F	Di	I	RR	X_1	X_2	X_3	X_4	X_5	X_6	W	C	
st p4 ➜ [p5+4]	F	Di	I	RR	X	SQ						C	
st p3 ➜ [p6+8]		F	Di	I	RR	X	SQ						C
ld [p7] ➜ p8		F	Di	I?	RR	X	M_1	M_2	W				C

- Can "ld [p7] ➜ p8" issue and begin execution?
  - Why or why not?
- If the load reads from either of the stores' addresses…
  - Load must get correct value, but stores don't write cache until commit…
- Solution: "memory forwarding"
  - Load also searches the Store Queue (in parallel with cache access)
  - Conceptually like register bypassing, but different implementation
    - Why? Addresses unknown until execute

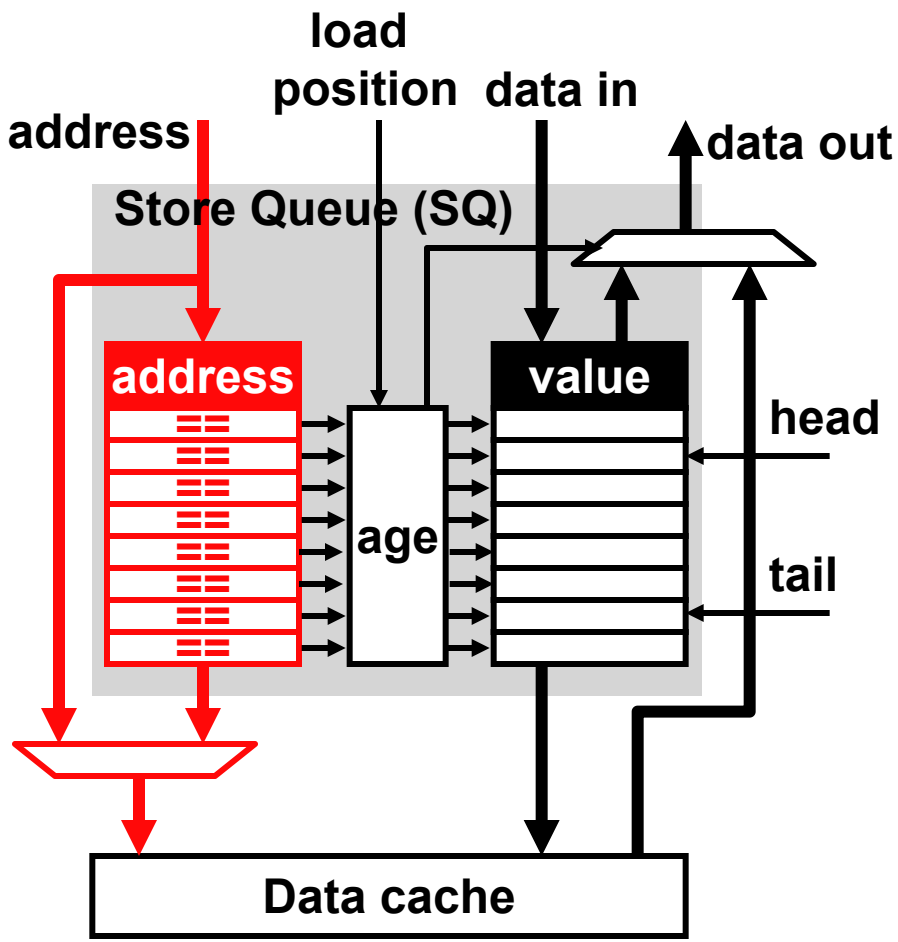
Problem #3: WAR Hazards

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 ➜ p3	F	Di	I	RR	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	RR	~~X~~	W	C		
ld [p3+4] ➜ p5		F	Di				I	RR	X	M_1	M_2	W	C
st p4 ➜ [p6+8]		F	Di	I	RR	X	SQ						C

- What if "p3+4 == p6 + 8"?
  - WAR: need to make sure that load doesn't read store's result
  - Need to get values based on "program order" not "execution order"
- Bad solution: require all stores/loads to execute in-order
- Good solution: add "age" fields to store queue (SQ)
  - Loads read from **youngest older matching** store
  - Another reason the SQ is a FIFO queue

Memory Forwarding via Store Queue

- Store Queue (SQ)
  - Holds all in-flight stores
  - CAM: searchable by address
  - Age logic: determine youngest matching store older than load
- Store rename/dispatch
  - Allocate entry in SQ
- Store execution
  - Update SQ
    - Address + Data
- Load execution
  - Search SQ identify youngest older matching store
    - Match? Read SQ
    - No Match? Read cache



When Can Loads Execute?

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 ➜ p3	F	Di	I	RR	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	RR	**X**	W	C		
st p5 ➜ [p3+4]		F	Di				I	RR	X	SQ	C		
ld [p6+8] ➜ p7		F	Di	**I?**	**RR**	**X**	**M_1**	**M_2**	**W**			C	

- Can "ld [p6+8] ➜ p7" issue in cycle 3
  - Why or why not?

When Can Loads Execute?

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 ➜ p3	F	Di	I	RR	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	RR	**X**	W	C		
st p5 ➜ [p3+4]		F	Di				I	RR	X	SQ	C		
ld [p6+8] ➜ p7		F	Di	**I?**	**RR**	**X**	**M_1**	**M_2**	**W**			C	

- Aliasing! Does p3+4 == p6+8?
  - If no, load should get value from memory
    - **Can it start to execute?**
  - If yes, load should get value from store
    - By reading the store queue?
    - **But the value isn't put into the store queue until cycle 9**
- **Key challenge**: don't know addresses until execution!
  - One solution: require all loads to wait for all earlier (prior) stores

Conservative Load Scheduling

- Conservative load scheduling:
  - All older stores have executed
    - Some architectures: split store address / store data
      - Only requires knowing addresses (not the store values)
  - Advantage: always safe
  - Disadvantage: performance (limits ILP)

Conservative Load Scheduling

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ld [p1] ➜ p4	F	Di	I	Rr	X	M_1	M_2	W	C							
ld [p2] ➜ p5	F	Di	I	Rr	X	M_1	M_2	W	C							
add p4, p5 ➜ p6		F	Di			I	Rr	X	W	C						
st p6 ➜ [p3]		F	Di				I	Rr	X	SQ	C					
ld [p1+4] ➜ p7			F	Di				I	Rr	X	M_1	M_2	W	C		
ld [p2+4] ➜ p8			F	Di				I	Rr	X	M_1	M_2	W	C		
add p7, p8 ➜ p9				F	Di						I	Rr	X	W	C	
st p9 ➜ [p3+4]				F	Di							I	Rr	X	SQ	C

Conservative load scheduling: can't issue ld [p1+4] until cycle 7!

Might as well be an in-order machine on this example

Can we do better? How?

Optimistic Load Scheduling

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ld [p1] ➜ p4	F	Di	I	Rr	X	M_1	M_2	W	C							
ld [p2] ➜ p5	F	Di	I	Rr	X	M_1	M_2	W	C							
add p4, p5 ➜ p6		F	Di			I	Rr	X	W	C						
st p6 ➜ [p3]		F	Di				I	Rr	X	SQ	C					
ld [p1+4] ➜ p7			F	Di	I	Rr	X	M_1	M_2	W	C					
ld [p2+4] ➜ p8			F	Di	I	Rr	X	M_1	M_2	W		C				
add p7, p8 ➜ p9				F	Di			I	Rr	X	W	C				
st p9 ➜ [p3+4]				F	Di				I	Rr	X	SQ	C			

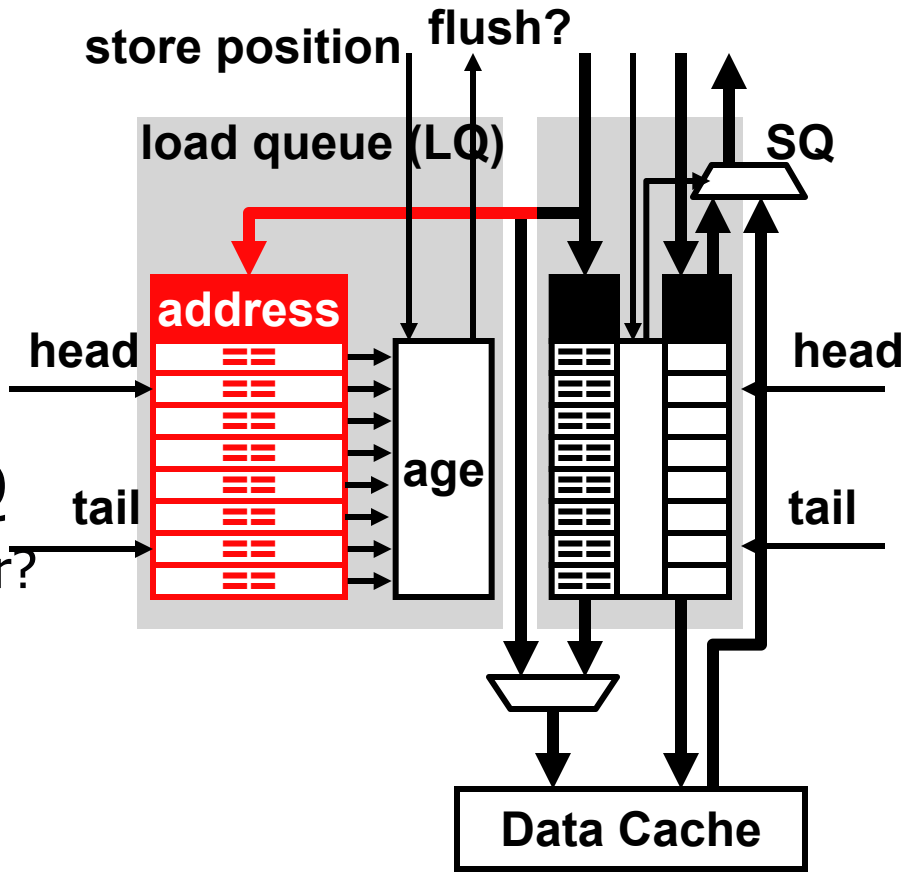
Optimistic load scheduling: can actually benefit from out-of-order!
Let's speculate!

Load Speculation

- Speculation requires three things…..
  - 1. When do we speculate?
  - 2. How do we detect a mis-speculation?
  - 3. How do we recover from mis-speculations?
    - **Squash offending load and all newer insns**
    - Similar to branch mis-prediction recovery

Load Queue

- Detects load ordering violations
- Load execution: Write address into LQ
  - Also note any store forwarded from
- Store execution: Search LQ
  - Younger load with same addr?
    - order violation, squash



Store Queue + Load Queue

- Store Queue: handles forwarding, allows OoO stores
  - Entry per store (allocated @ dispatch, deallocated @ commit)
  - Written by stores (@ execute)
  - Searched by loads (@ execute)
  - Read from SQ to write data cache (@ commit)
- Load Queue: detects ordering violations
  - Entry per load (allocated @ dispatch, deallocated @ commit)
  - Written by loads (@ execute)
  - Searched by stores (@ execute)
- Both together
  - Allows aggressive load scheduling
  - Stores don’t constrain load execution

Optimistic Load Scheduling Problem

- Allows loads to issue before older stores
  - Increases ILP
  - \+ Good: When no conflict, increases performance
  - \- Bad: Conflict => squash => worse performance than waiting

- Can we have our cake AND eat it too?

Predictive Load Scheduling

- Predict which loads must wait for stores

- Fool me once, shame on you-- fool me twice?
  - Loads default to aggressive
  - Keep table of load PCs that have been caused squashes
    - Schedule these conservatively
  - \+ Simple predictor
  - \- Makes "bad" loads wait for **all** older stores

- More complex predictors used in practice
  - Predict which stores loads should wait for
  - "Store Sets" paper

Load/Store Queue Examples

Initial State

(Stores to different addresses)

1. St p1 → [p2]
2. St p3 → [p4]
3. Ld [p5] → p6

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val

Cache

Addr	Val
100	13
200	17

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val

Cache

Addr	Val
100	13
200	17

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val

Cache

Addr	Val
100	13
200	17

Good Interleaving

(Shows importance of address check)

1. St p1 ➜ [p2]
2. St p3 ➜ [p4]
3. Ld [p5] ➜ p6

1. St p1 ➜ [p2]

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1	100	5

Cache

Addr	Val
100	13
200	17

2. St p3 ➜ [p4]

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1	100	5
2	200	9

Cache

Addr	Val
100	13
200	17

3. Ld [p5] ➜ p6

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	**5**
p7	---
p8	---

Load Queue

Bdy	Addr
3	100

Store Queue

Bdy	Addr	Val
1	100	5
2	200	9

Cache

Addr	Val
100	13
200	17

Different Initial State

(All to same address)

1. St p1 → [p2]
2. St p3 → [p4]
3. Ld [p5] → p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val

Cache

Addr	Val
100	13
200	17

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val

Cache

Addr	Val
100	13
200	17

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val

Cache

Addr	Val
100	13
200	17

Good Interleaving #1

(Program Order)

1. St p1 ➜ [p2]
2. St p3 ➜ [p4]
3. Ld [p5] ➜ p6

1. St p1 ➜ [p2]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1	100	5

Cache

Addr	Val
100	13
200	17

2. St p3 ➜ [p4]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

3. Ld [p5] ➜ p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	**9**
p7	---
p8	---

Load Queue

Bdy	Addr
3	100

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

Good Interleaving #2

(Stores reordered)

1. St p1 ➜ [p2]
2. St p3 ➜ [p4]
3. Ld [p5] ➜ p6

2. St p3 ➜ [p4]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1		
2	100	9

Cache

Addr	Val
100	13
200	17

1. St p1 ➜ [p2]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

3. Ld [p5] ➜ p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	**9**
p7	---
p8	---

Load Queue

Bdy	Addr
3	100

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

Bad Interleaving #1

(Load reads the cache)

1. St p1 ➜ [p2]
2. St p3 ➜ [p4]
3. Ld [p5] ➜ p6

3. Ld [p5] ➜ p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	13
p7	---
p8	---

Load Queue

Bdy	Addr
3	100

Store Queue

Bdy	Addr	Val
1		
2		

Cache

Addr	Val
100	13
200	17

2. St p3 ➜ [p4]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	13
p7	---
p8	---

Load Queue

Bdy	Addr
3	100

Store Queue

Bdy	Addr	Val
2	100	9

Cache

Addr	Val
100	13
200	17

Bad Interleaving #2

(Load gets value from wrong store)

1. St p1 ➜ [p2]
2. St p3 ➜ [p4]
3. Ld [p5] ➜ p6

1. St p1 ➜ [p2]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store

Store Queue

Bdy	Addr	Val
1	100	5

Cache

Addr	Val
100	13
200	17

3. Ld [p5] ➜ p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	**5**
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store
3	100	1

Store Queue

Bdy	Addr	Val
1	100	5
2		

Cache

Addr	Val
100	13
200	17

2. St p3 ➜ [p4]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	5
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store
3	100	1

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

Bad/Good Interleaving

(Load gets value from correct store, but does it work?)

1. St p1 ➜ [p2]
2. St p3 ➜ [p4]
3. Ld [p5] ➜ p6

2. St p3 ➜ [p4]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store

Store Queue

Bdy	Addr	Val
1		
2	100	9

Cache

Addr	Val
100	13
200	17

3. Ld [p5] ➜ p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	**9**
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store
3	100	2

Store Queue

Bdy	Addr	Val
1		
2	100	9

Cache

Addr	Val
100	13
200	17

1. St p1 ➜ [p2]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	9
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store
3	100	2

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

Out-of-Order Everywhere

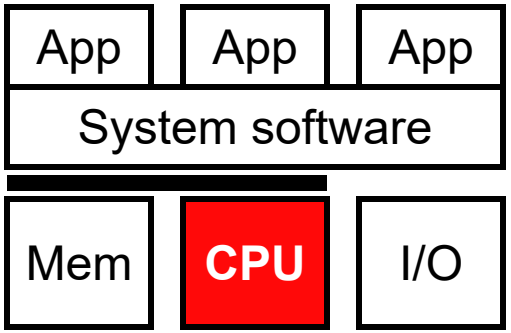
OoO Everywhere

- iPhone 7
  - Apple A10 processor
  - details very closely guarded
  - definitely OoO
  - some things known about Cyclone (Apple A7 chip)
    - 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

OoO everywhere for a while now

- Qualcomm Krait 400 processor
  - based on ARM Cortex A15 processor
  - **out-of-order** 2.5GHz quad-core
  - 3-wide fetch/decode
  - 4-wide issue
  - 11-stage integer pipeline
  - 28nm process technology
  - 4/4KB DM L1$, 16/16KB 4-way SA L2$, 2MB 8-way SA L3$

Summary: Scheduling



- Code scheduling
  - To reduce pipeline stalls
  - To increase ILP (insn-level parallelism)
- Static scheduling by the compiler
  - Approach & limitations
- Dynamic scheduling in hardware
  - Register renaming
  - Instruction selection
  - Handling memory operations
- Up next: multicore