332
Advanced Computer Architecture
Chapter 2: part 2

Dynamic scheduling, out-of-order execution, register renaming ***with speculative execution***

Hennessy and Patterson 6th ed Section 3.6 pp208-217 and pp234-238

October 2022
Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (4-6th ed), and on the lecture slides of David Patterson's Berkeley course (CS252)*

Course materials online on
https://scientia.doc.ic.ac.uk/2223/modules/60001/materials and
https://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/aca20/

What about Precise Interrupts?

- Tomasulo had:

  In-order issue, out-of-order execution, and out-of-order completion

- Need to “fix” the out-of-order completion aspect so that we can find precise breakpoint in instruction stream
  - Suppose we have a page fault or a divide-by-zero exception?

- Actually we have the same issue with **branch speculation**…

- **The answer: add a stage that “commits” the state**
- **In issue order**

Four Steps of the *Speculative* Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

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

2. Execution—operate on operands (EX)

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

3. Write result—finish execution (WB)

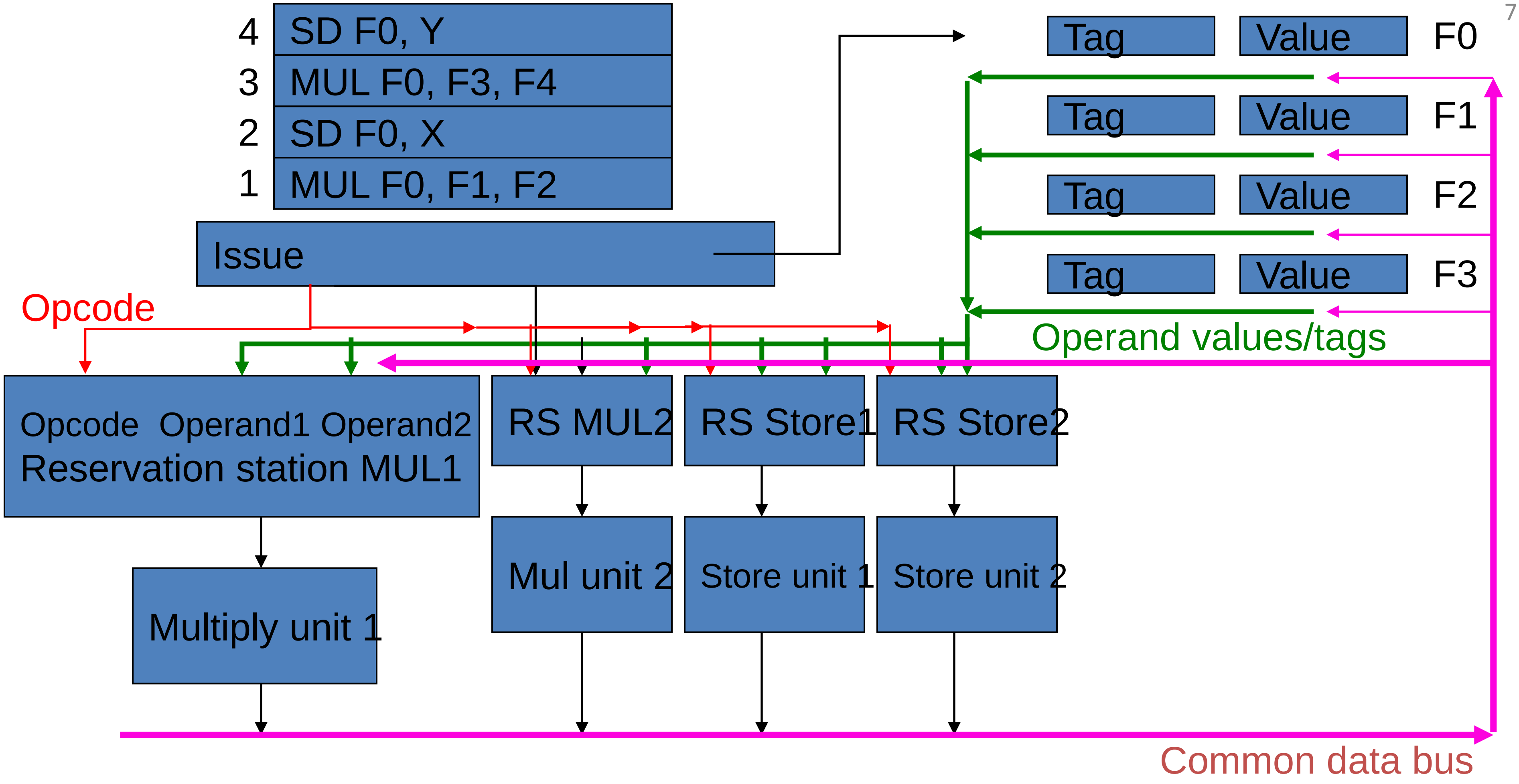
    Write on Common Data Bus to all awaiting FUs
    & reorder buffer; mark reservation station available.

4. Commit—update register with reorder result

    When an instruction is at the head of reorder buffer, *and* its result is present:

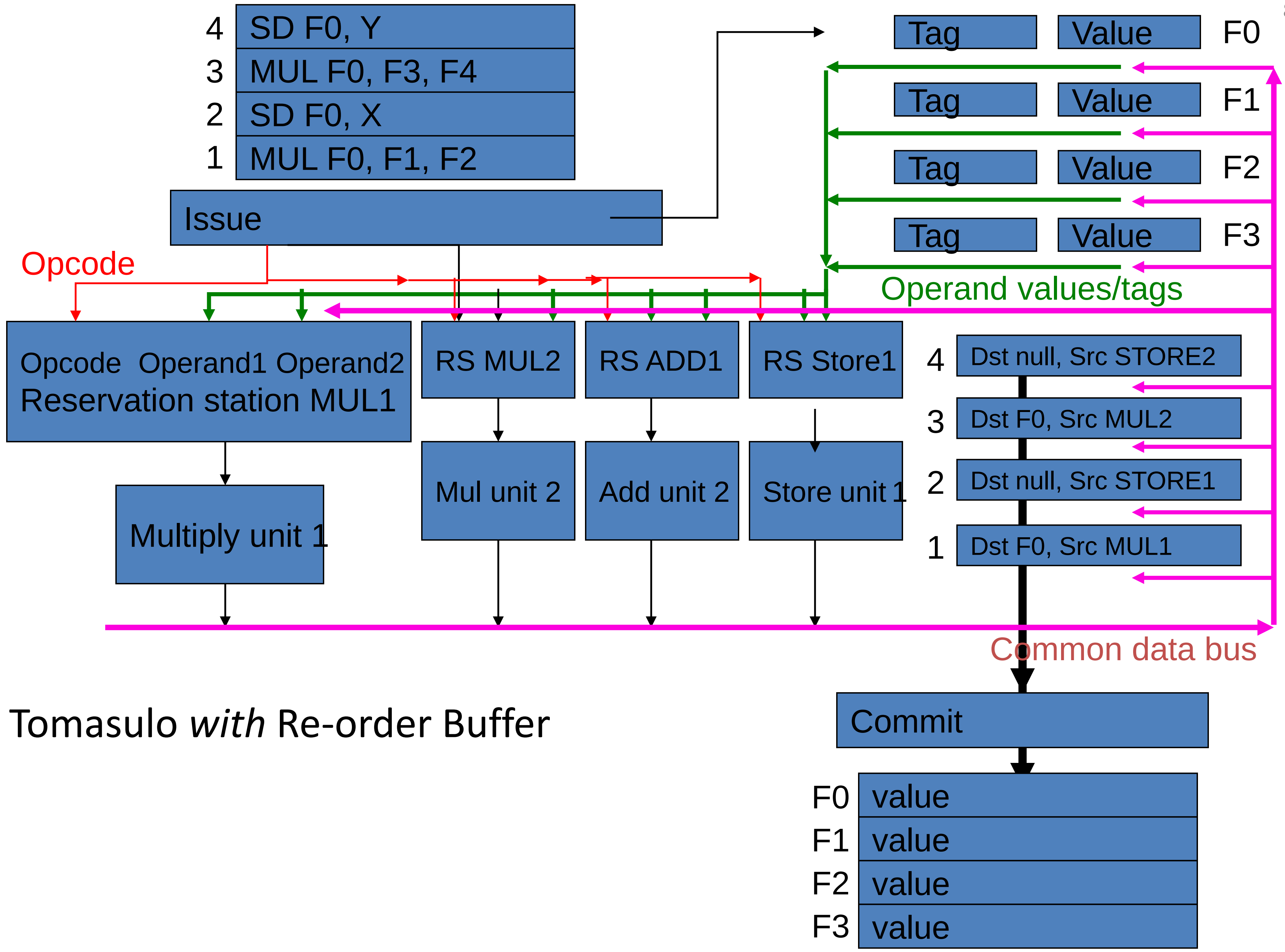
        update the (commit-side) register with the result (or store to memory), and remove the instruction from the reorder buffer.

Mispredicted branch flushes reorder buffer

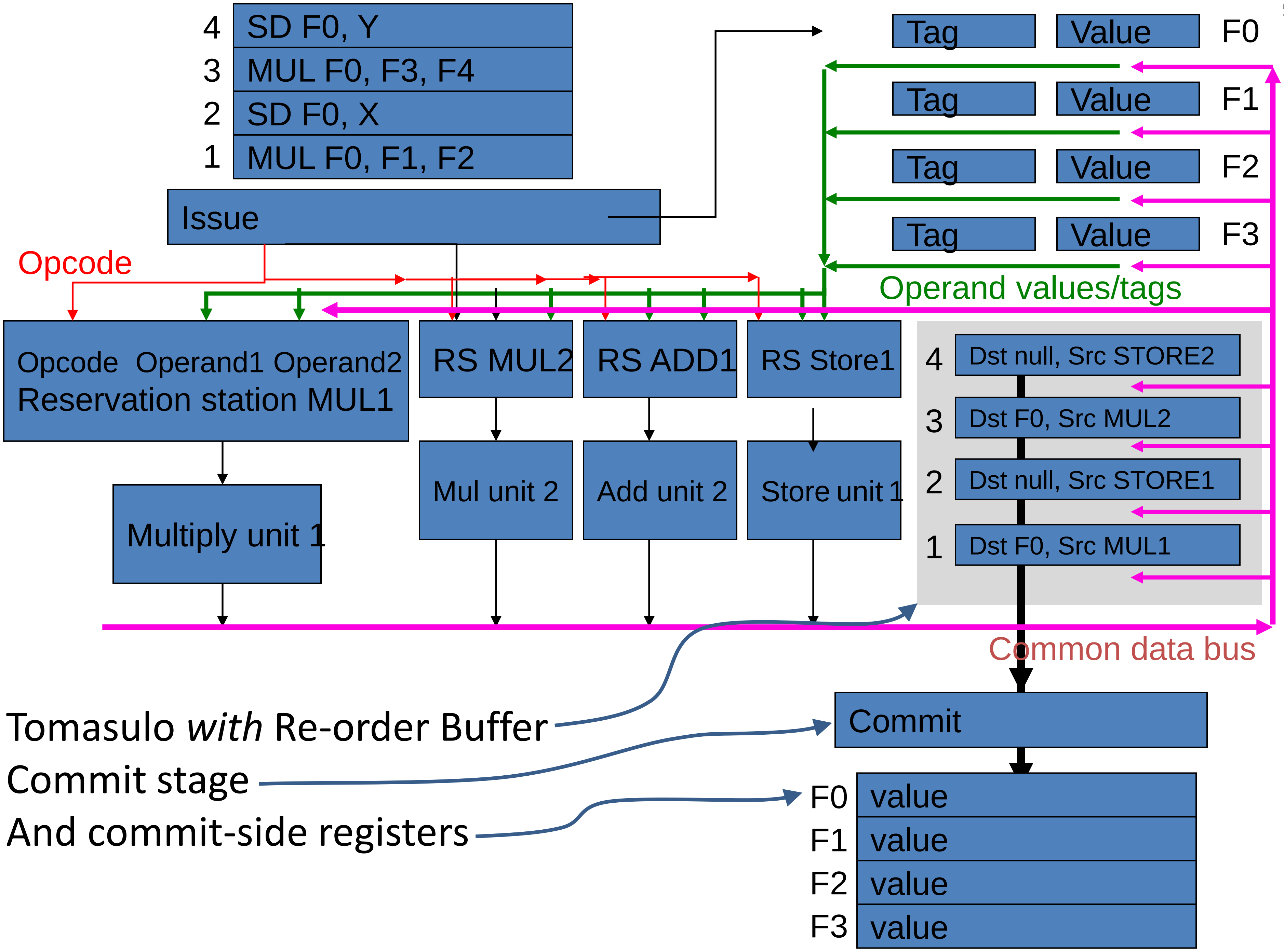


Tomasulo *without* Re-order Buffer

(from previous lecture)



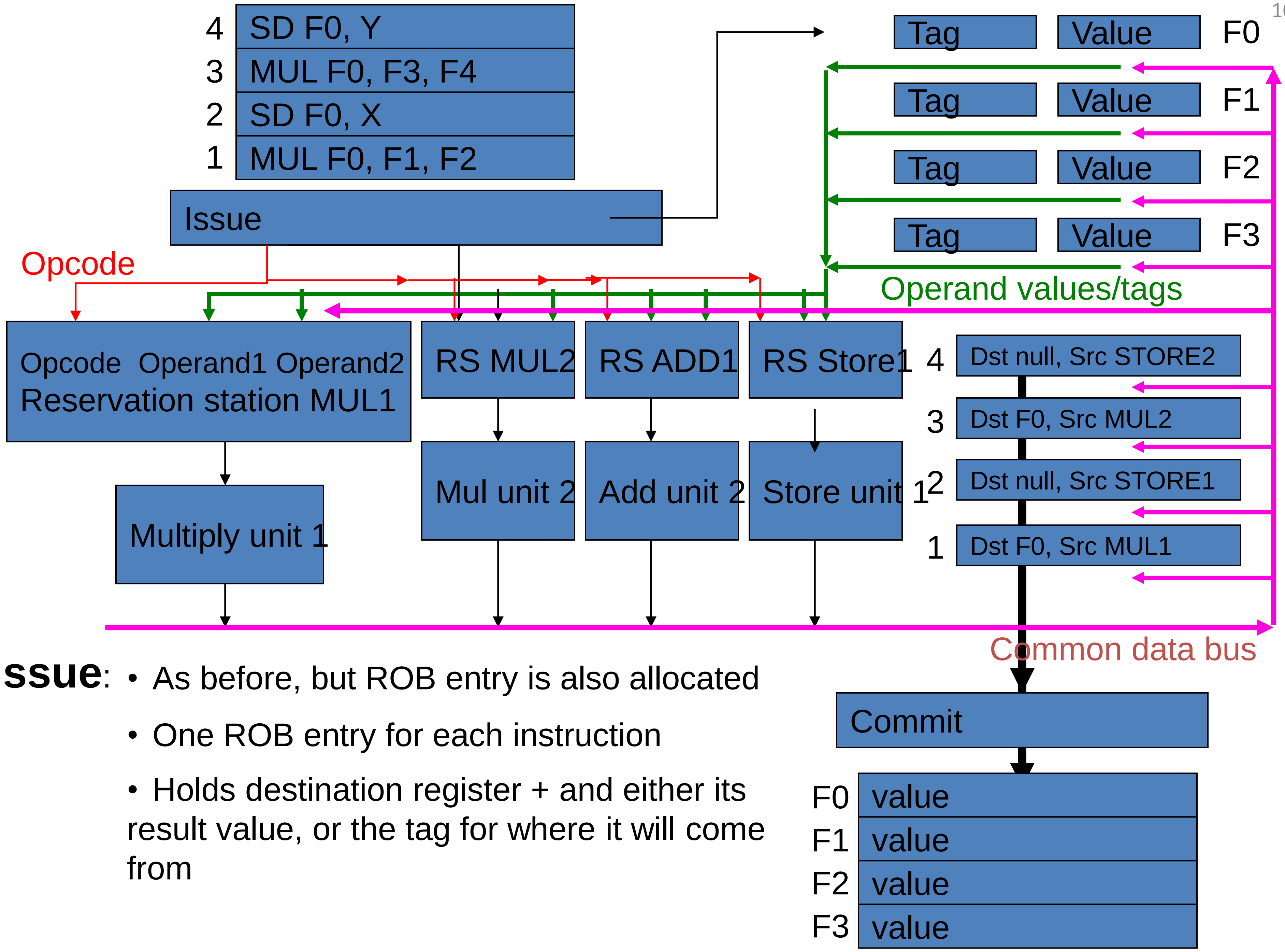
Tomasulo *with* Re-order Buffer



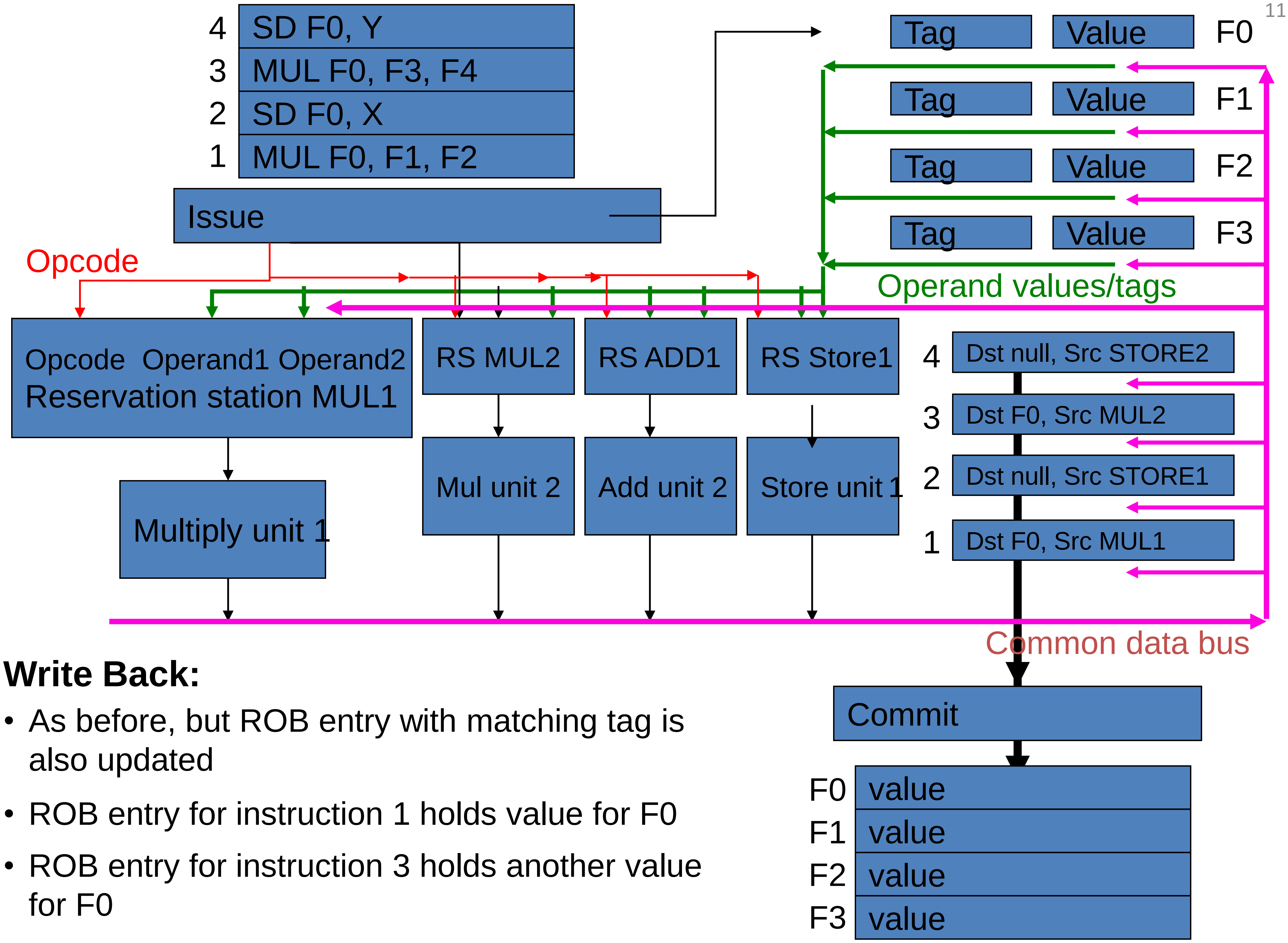
Tomasulo *with* Re-order Buffer

Commit stage

And commit-side registers

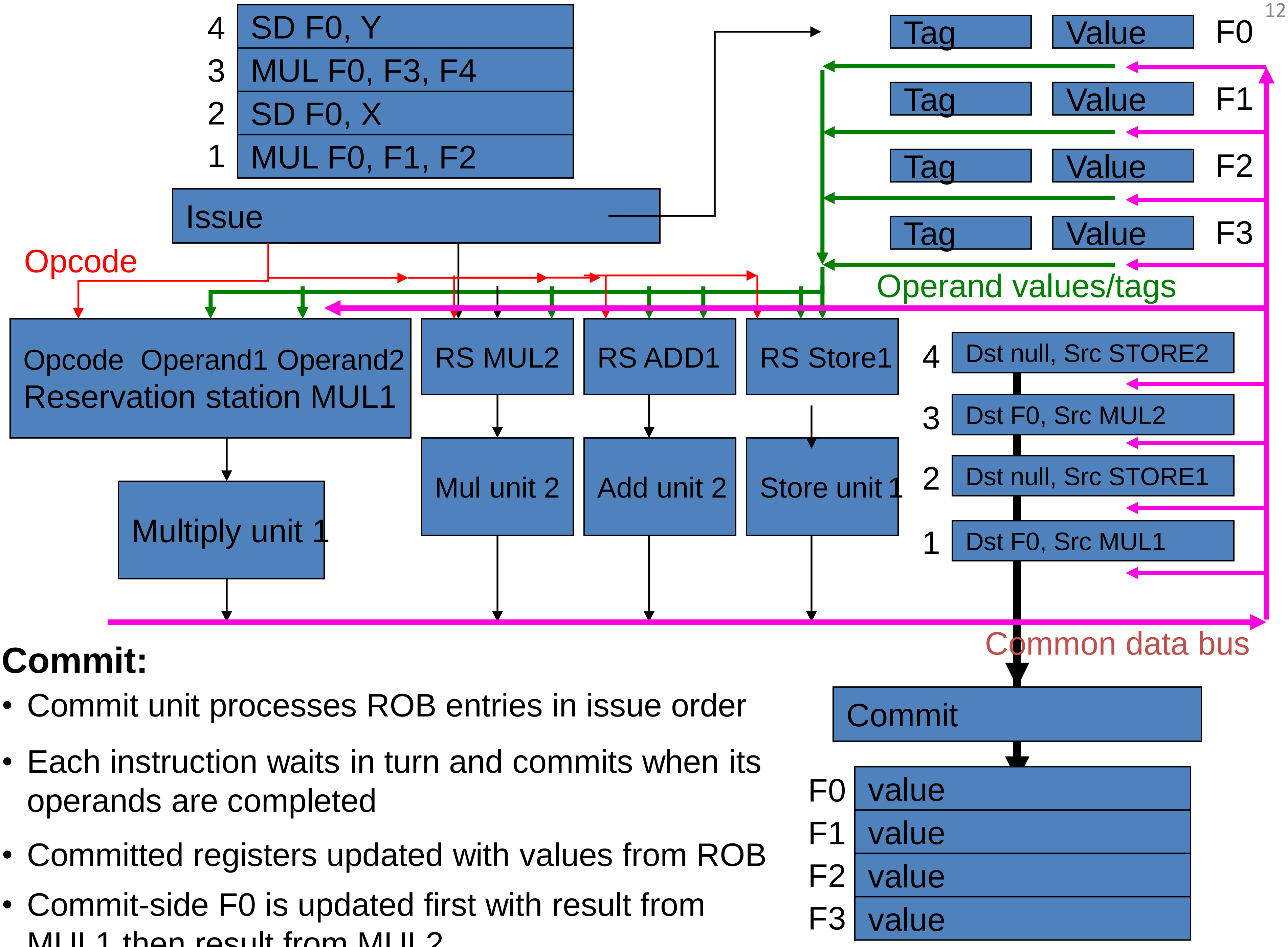


Issue:
- As before, but ROB entry is also allocated
- One ROB entry for each instruction
- Holds destination register + and either its result value, or the tag for where it will come from



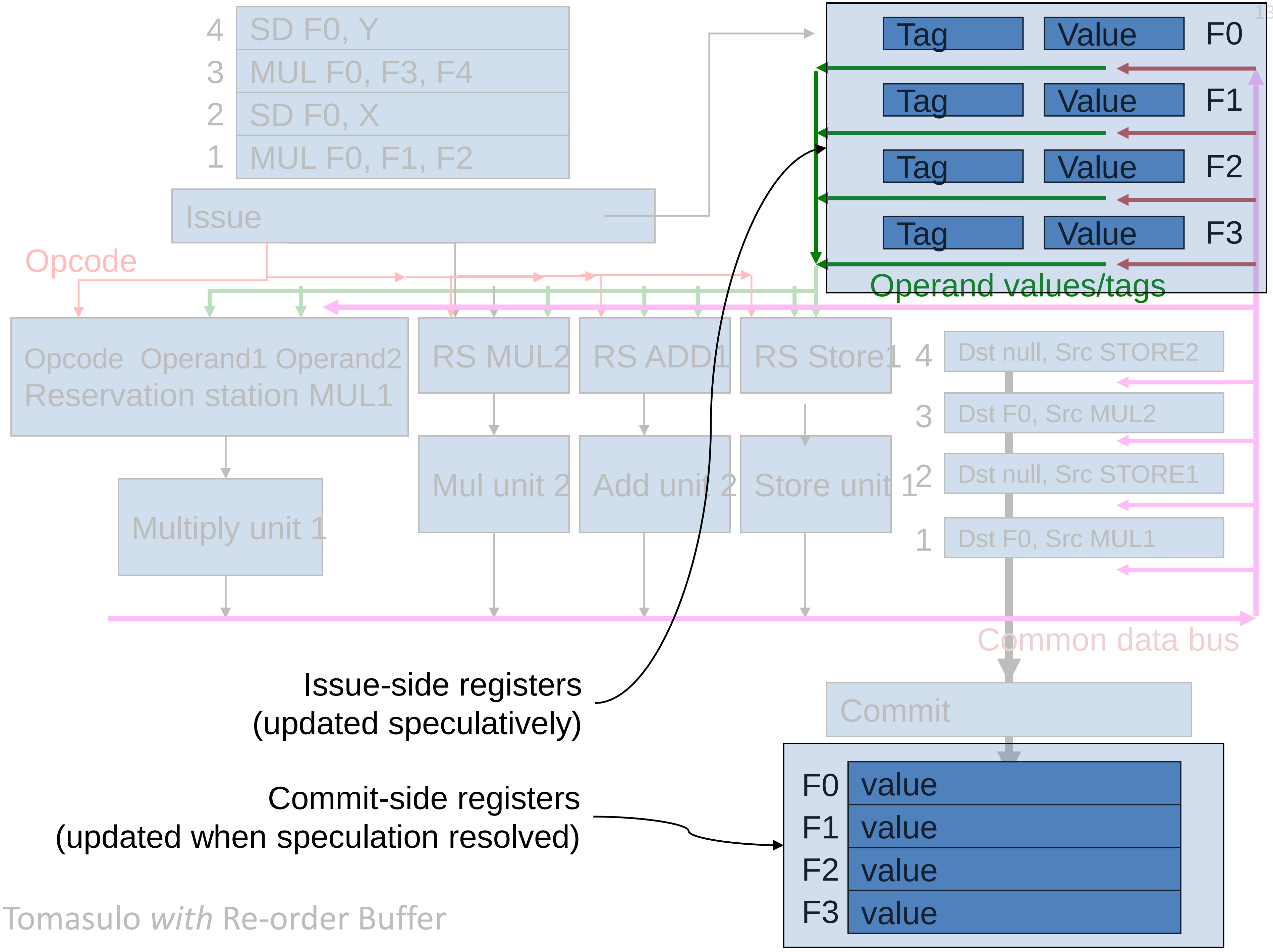
Write Back:

- As before, but ROB entry with matching tag is also updated
- ROB entry for instruction 1 holds value for F0
- ROB entry for instruction 3 holds another value for F0

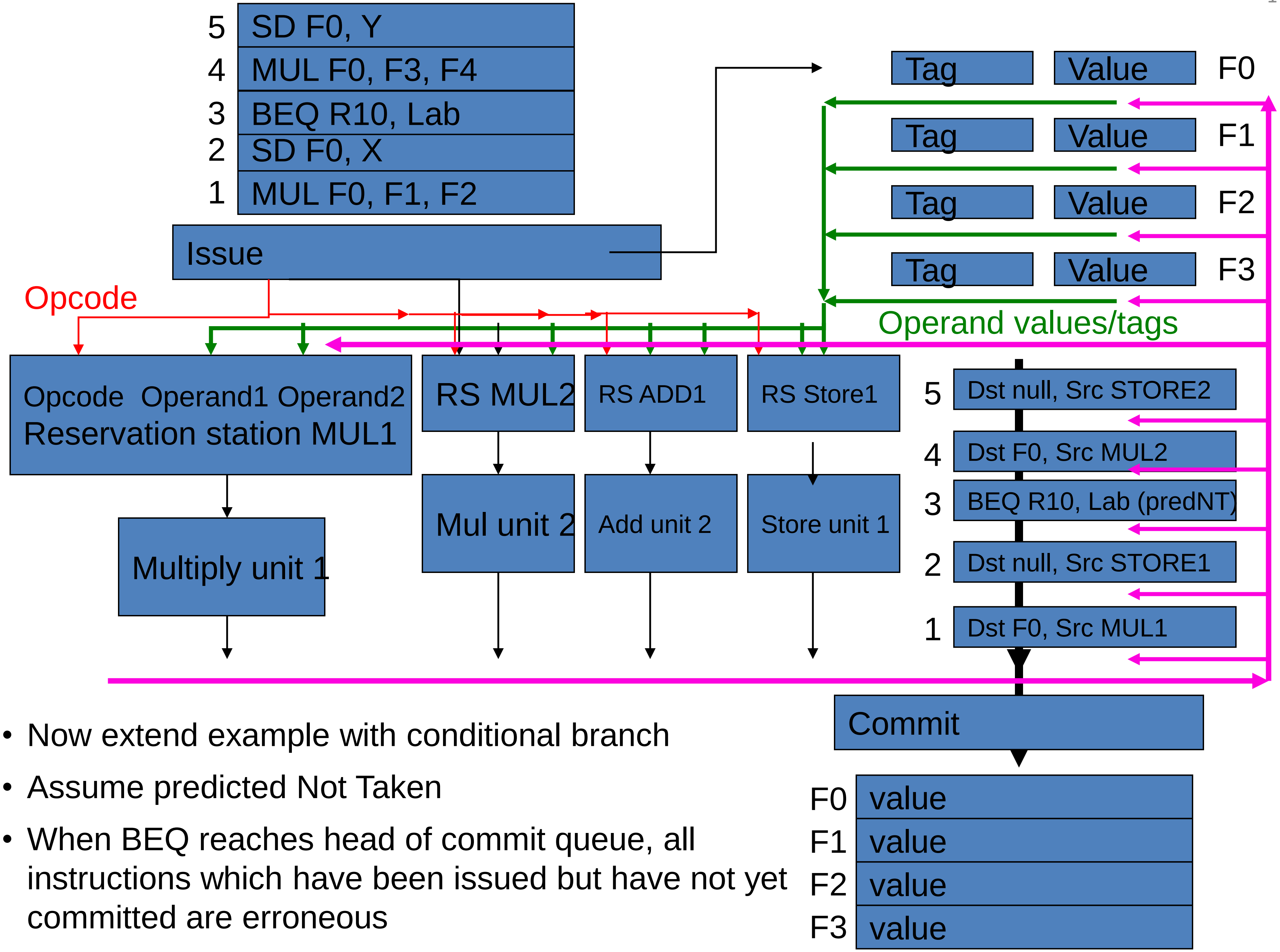


Commit:

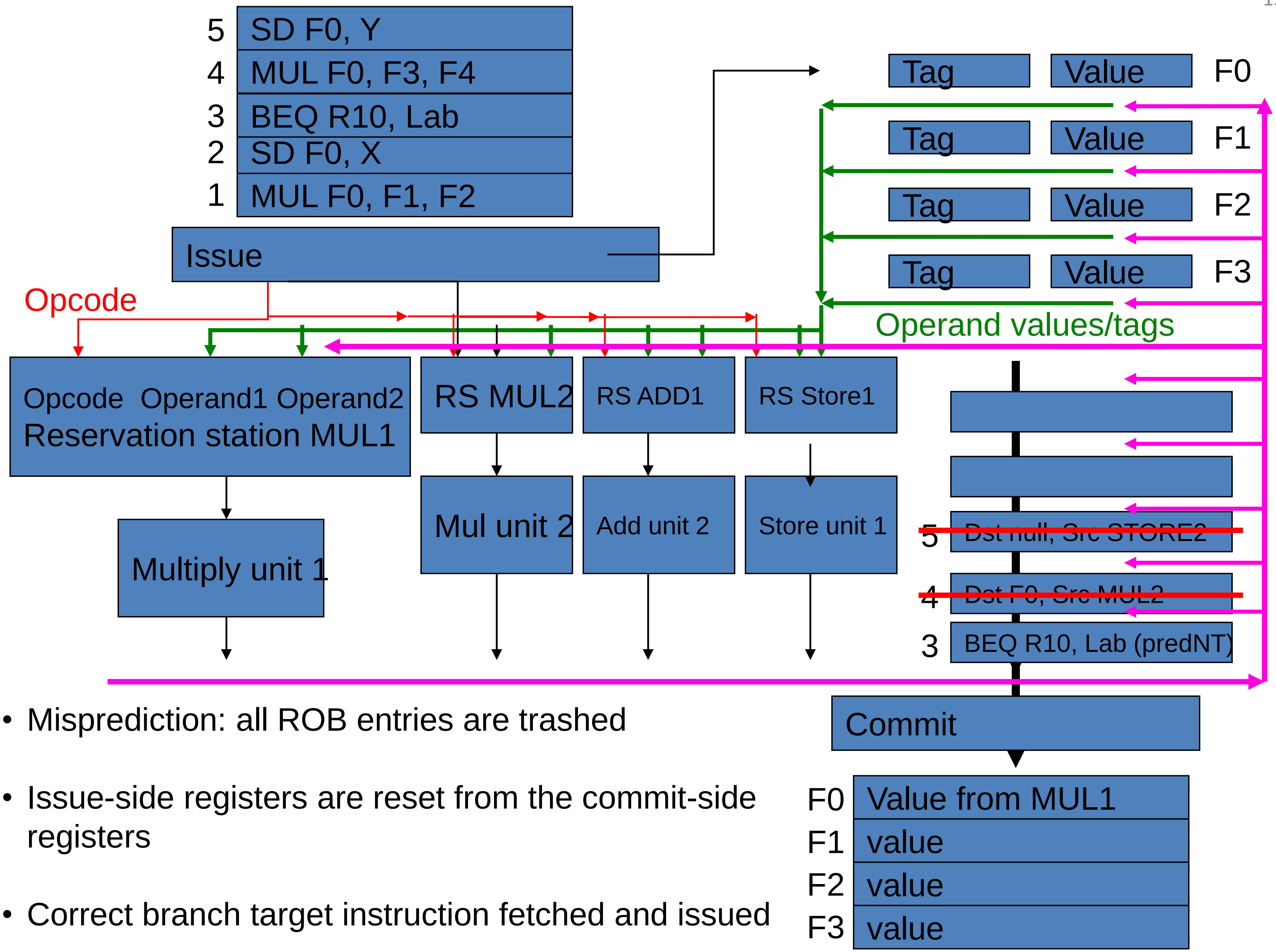
- Commit unit processes ROB entries in issue order
- Each instruction waits in turn and commits when its operands are completed
- Committed registers updated with values from ROB
- Commit-side F0 is updated first with result from MUL1 then result from MUL2



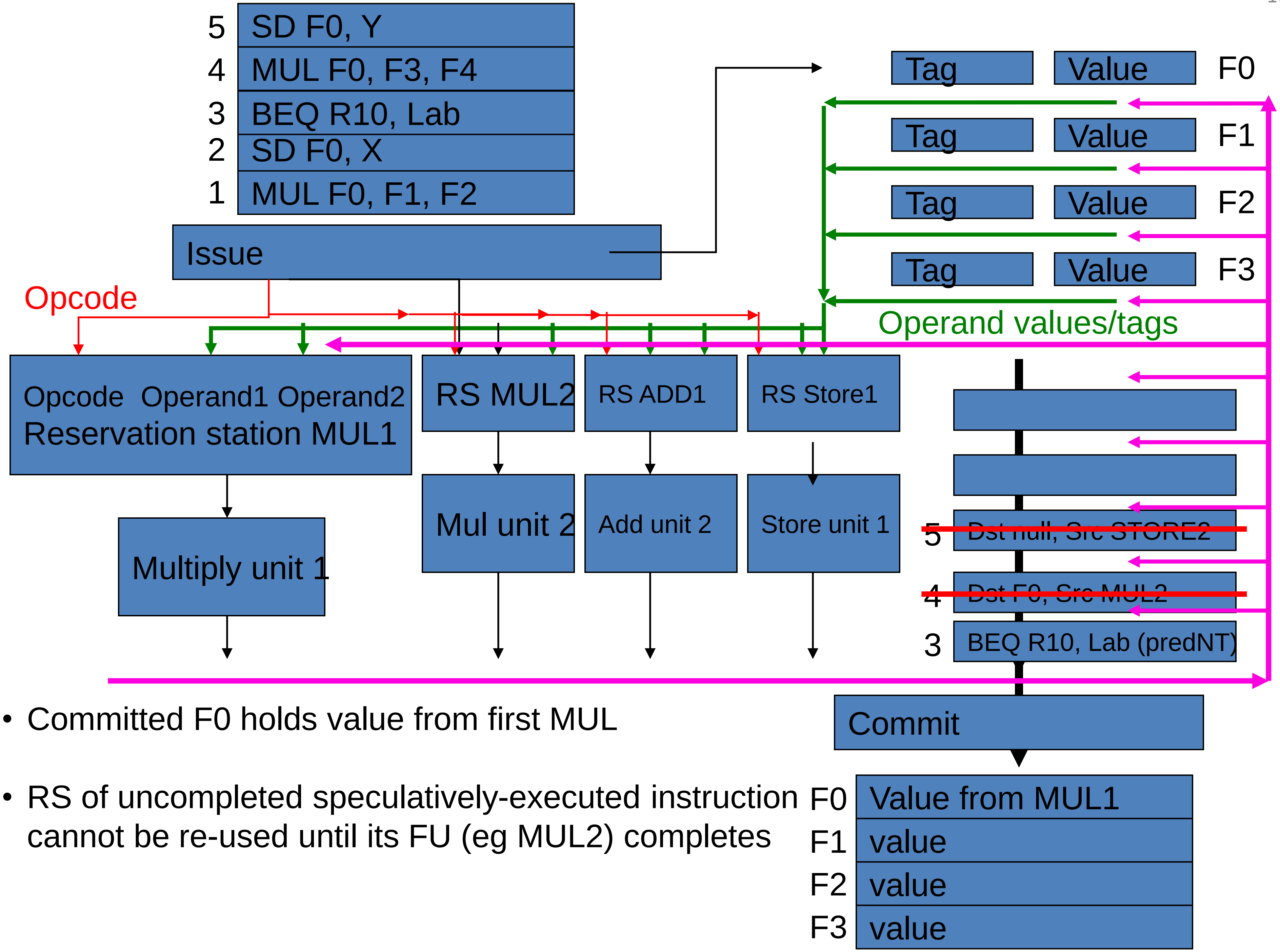
Tomasulo *with* Re-order Buffer



- Now extend example with conditional branch
- Assume predicted Not Taken
- When BEQ reaches head of commit queue, all instructions which have been issued but have not yet committed are erroneous



- Misprediction: all ROB entries are trashed
- Issue-side registers are reset from the commit-side registers
- Correct branch target instruction fetched and issued



- Committed F0 holds value from first MUL
- RS of uncompleted speculatively-executed instruction cannot be re-used until its FU (eg MUL2) completes

Some subtleties to think about…

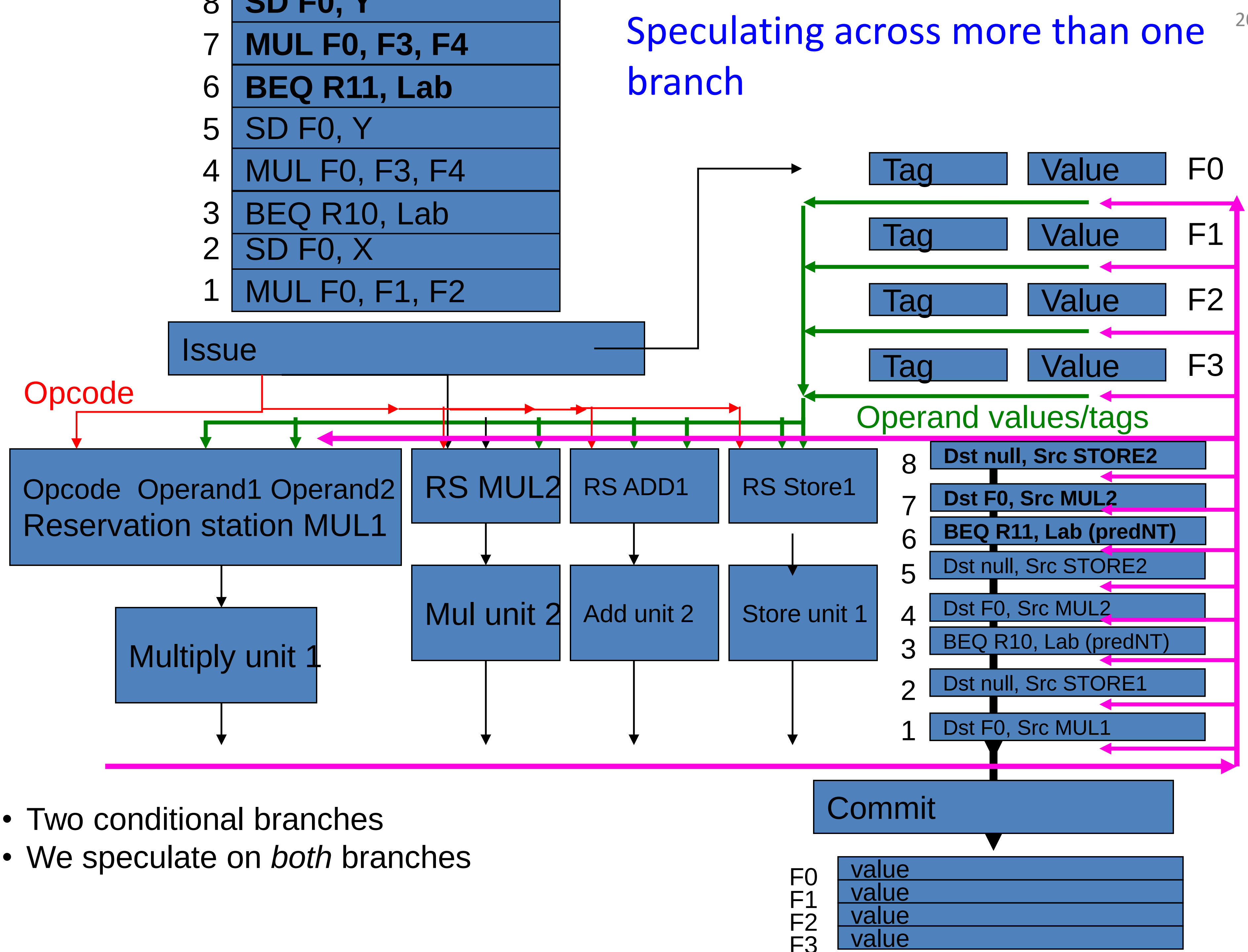
- It’s vital to reduce the branch misprediction penalty. Does the Tomasulo+ROB scheme described here roll-back as soon as the branch is found to be mispredicted?

- This discussion has assumed a single-issue machine. How can these ideas be extended to allow multiple instructions to be issued per cycle?
  - Issue
  - Monitoring CDBs for completion
  - Handling multiple commits per cycle

Some subtleties to think about…

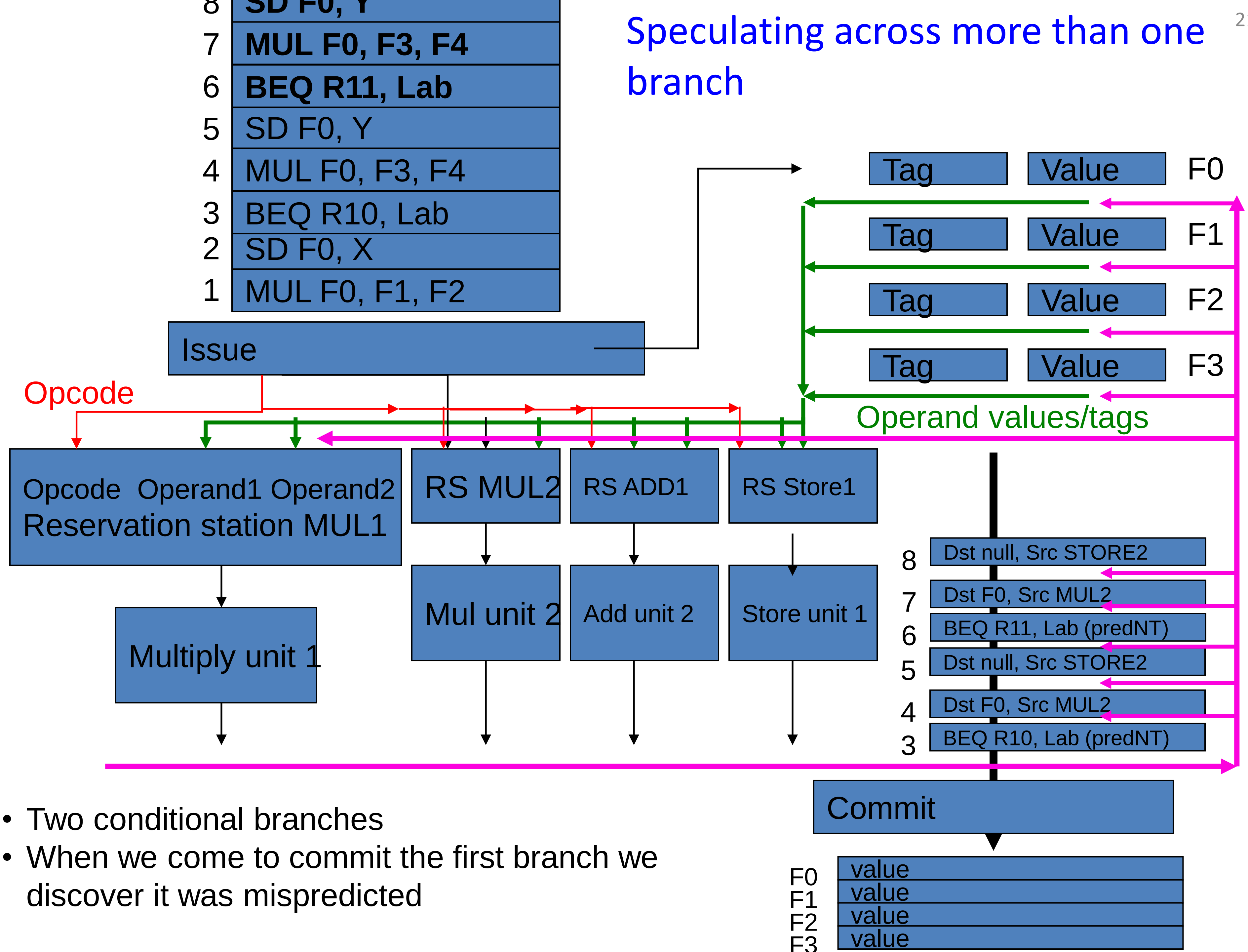
- What if a second conditional branch is encountered, before the outcome of the first is resolved?

Speculating across more than one branch



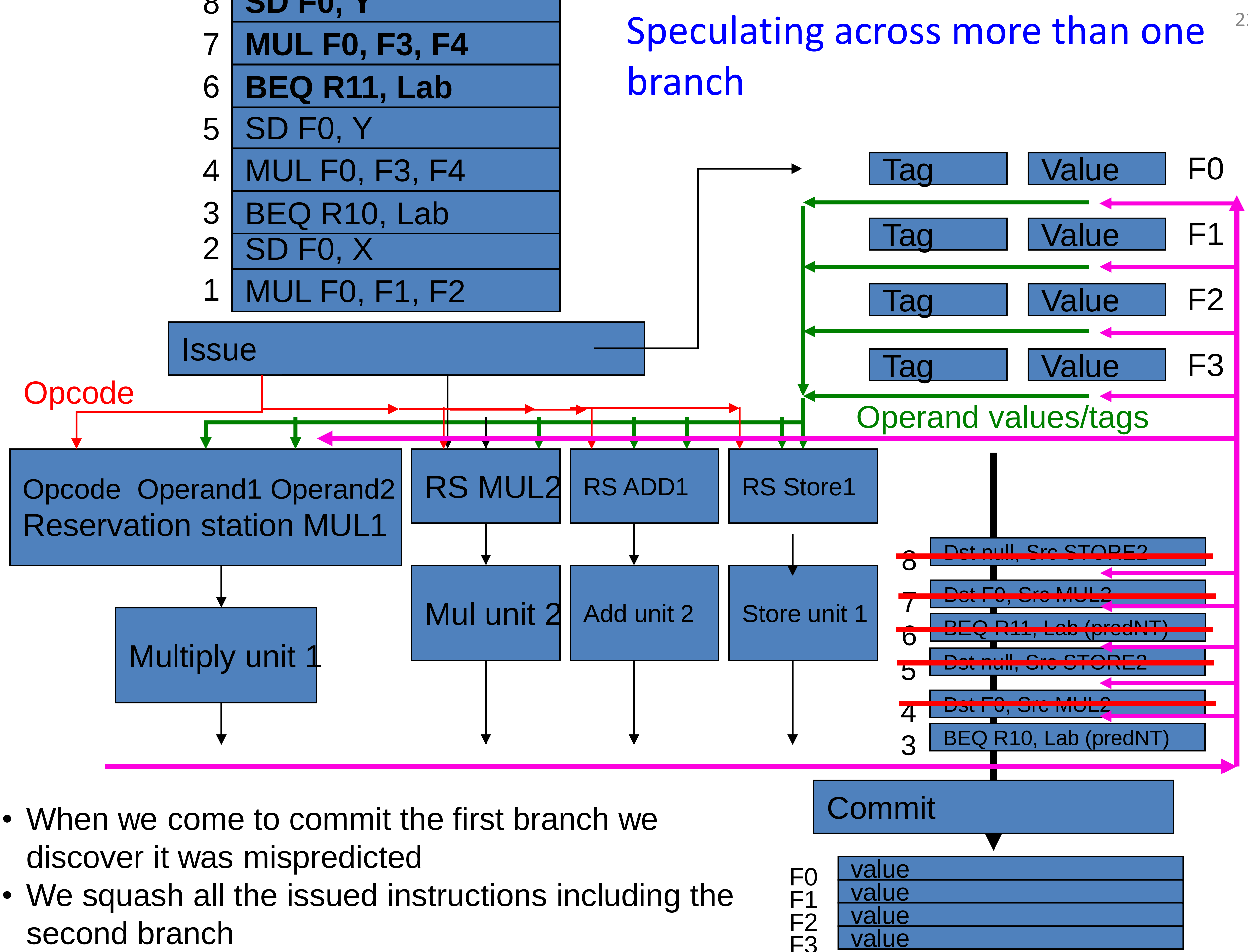
- Two conditional branches
- We speculate on *both* branches

Speculating across more than one branch



- Two conditional branches
- When we come to commit the first branch we discover it was mispredicted

Speculating across more than one branch



- When we come to commit the first branch we discover it was mispredicted
- We squash all the issued instructions including the second branch

Some subtleties to think about…

- Stores are buffered in the ROB, and committed only when the instruction is committed.
- A load can be issued while several stores (perhaps to the same address) are uncommitted. We need to make sure the load gets the right data. See:

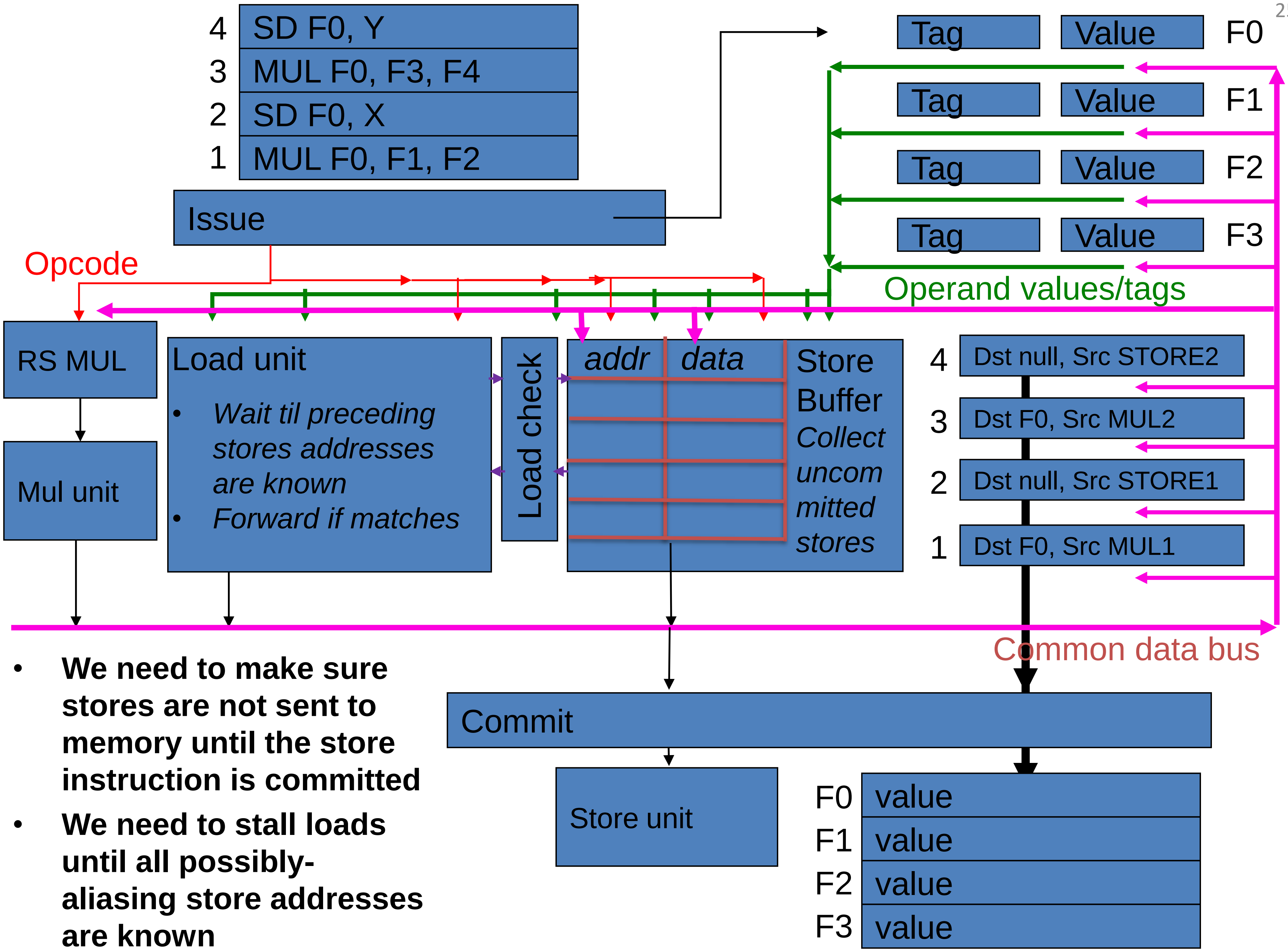
  Shen and Lipasti "Modern Processor Design" pg 271, or

  http://home.eng.iastate.edu/~zzhang/courses/cpre585_f03/slides/lecture11.pdf

- *This lies beyond the depth we have time to cover properly in this course, but let's look at some of the issues*

Stores and loads with speculation

- We need to make sure stores are not sent to memory until the store instruction is committed
- We need to stall loads until all preceding stores have committed
  - ?
  - Or: until all possibly-aliasing stores have committed?
  - Or: until the addresses of all preceding uncommitted stores have been determined
- If/when the addresses of a load and all preceding uncommitted stores are known…
  - And if none of the store addresses match the load
  - Then the load can proceed
  - If the address of the load matches the address of an uncommitted store, we can forward the store's data to the load



- **We need to make sure stores are not sent to memory until the store instruction is committed**
- **We need to stall loads until all possibly-aliasing store addresses are known**

Store-to-load forwarding

- **The Tomasulo scheme works on *registers* – it derives dependences between register-register instructions**
- **The registers being used are always known at issue time**
- **Loads and stores use *computed* addresses, which may or may *not* be known at issue time – consider:**

```
i1   SD F0 0(R3)     // store F0 at address R3
i2   LD R2 0(R1)     // load an address from memory
i3   SD F1 0(R2)     // store F1 to that address
i4   LD F2 0(R3)     // load F1 from address R3
```

- **Can we (should we?) forward F0 from *i1* to *i4*?**
- **What if R1=R3?**
- **We could wait (as shown in previous slide)**
- **We could speculate! And then check for the misprediction**
- **We could add a forwarding predictor, to improve the speculation**

Store-to-load forwarding

- Memory dependence *speculation* is the idea that we might allow a load to proceed* before we know for sure which, if any, prior uncommitted store instruction writes to its address**.
- (* proceed either by forwarding a value from some store whose *value* is known, or proceed by going to memory)
- (** we may know the load's address but not (all) the addresses of the older stores. We might not know the load's address)
- Memory dependence speculation is when we use a predictor to decide when to do this.
- See Memory dependence prediction - Wikipedia
- I think this article (start at page 8) is particularly clear:
- https://www.jilp.org/vol2/v2paper13.pdf

Design alternatives for o-o-o processor architectures

- See:
  - The Microarchitecture of the Pentium 4 Processor (Hinton et al, Intel Tech Jnl Q1 2001)
  - The SimpleScalar Tool Set, Version 2.0 (Burger and Austin, http://www.simplescalar.com/docs/users_guide_v2.pdf)
  - Wattch: a framework for architectural-level power analysis and optimizations (Brooks et al, ISCA 2000) *www.tortolaproject.com/papers/brooks00wattch.pdf*

- *Specifically:*
  - *Register Update Unit (RUU, as in Simplescalar) versus Re-Order Buffer*
  - *Realisation in Pentium III and Pentium 4 ("Netburst")*
    - *Frontend and Retirement Register Alias Tables (RATs)*

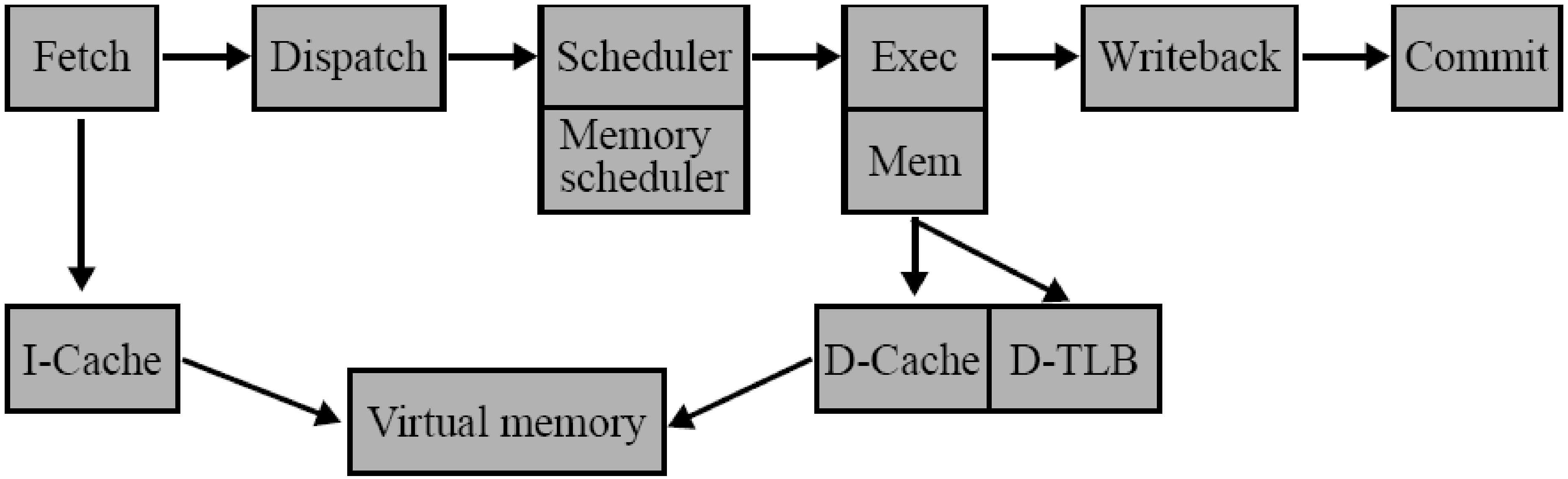
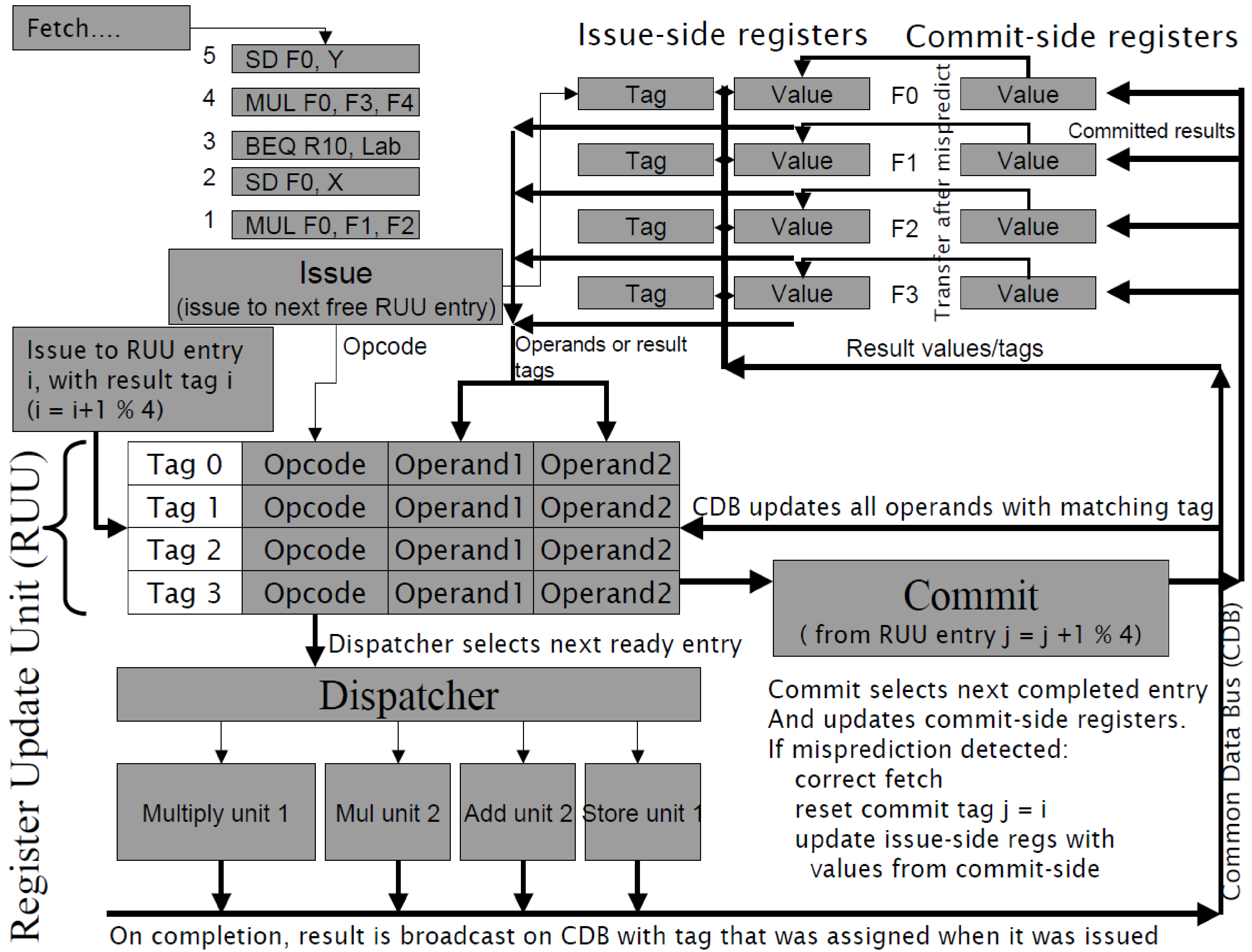


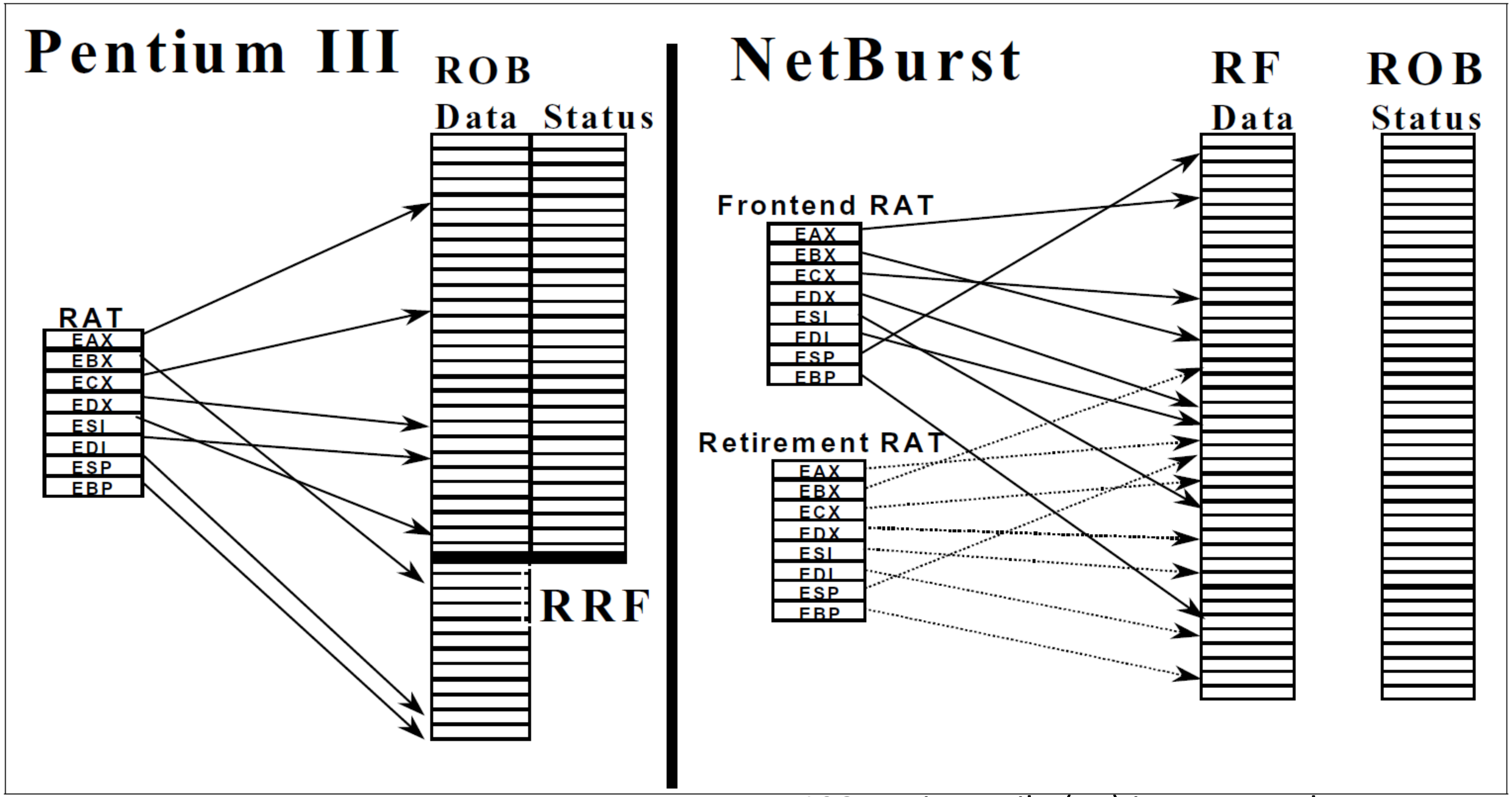
Figure 5. Pipeline for sim-outorder

- Simplescalar is a software simulation of a processor microarchitecture
- It simulates a multi-issue out-of-order design with speculative execution
- Many aspects of the design can be controlled by parameters
- Simplescalar uses a Register Update Unit, which combines ROB and reservation stations in a single pool


Fetch....
5 SD F0, Y
4 MUL F0, F3, F4
3 BEQ R10, Lab
2 SD F0, X
1 MUL F0, F1, F2
Issue-side registers
Commit-side registers
Tag
Value
F0
F1
F2
F3
Committed results
Transfer after mispredict
Issue
(issue to next free RUU entry)
Issue to RUU entry i, with result tag i (i = i+1 % 4)
Opcode
Operands or result tags
Result values/tags
Register Update Unit (RUU)
Tag 0
Tag 1
Tag 2
Tag 3
Operand1
Operand2
CDB updates all operands with matching tag
Commit
(from RUU entry j = j +1 % 4)
Dispatcher selects next ready entry
Dispatcher
Commit selects next completed entry
And updates commit-side registers.
If misprediction detected:
correct fetch
reset commit tag j = i
update issue-side regs with values from commit-side
Multiply unit 1
Mul unit 2
Add unit 2
Store unit 1
Common Data Bus (CDB)
On completion, result is broadcast on CDB with tag that was assigned when it was issued

RUU vs ROB

- In the Tomasulo+ROB design shown in these slides, registers *and* ROB entries have a tag
  - Every register, ROB entry and reservation station needs a comparator to monitor the CDB
- With the RUU, the tags *are* the ROB entry numbers
  - So the ROB is *indexed* by the tag on the CDB
  - The ROB entry serves are a renamed register for its instruction’s result



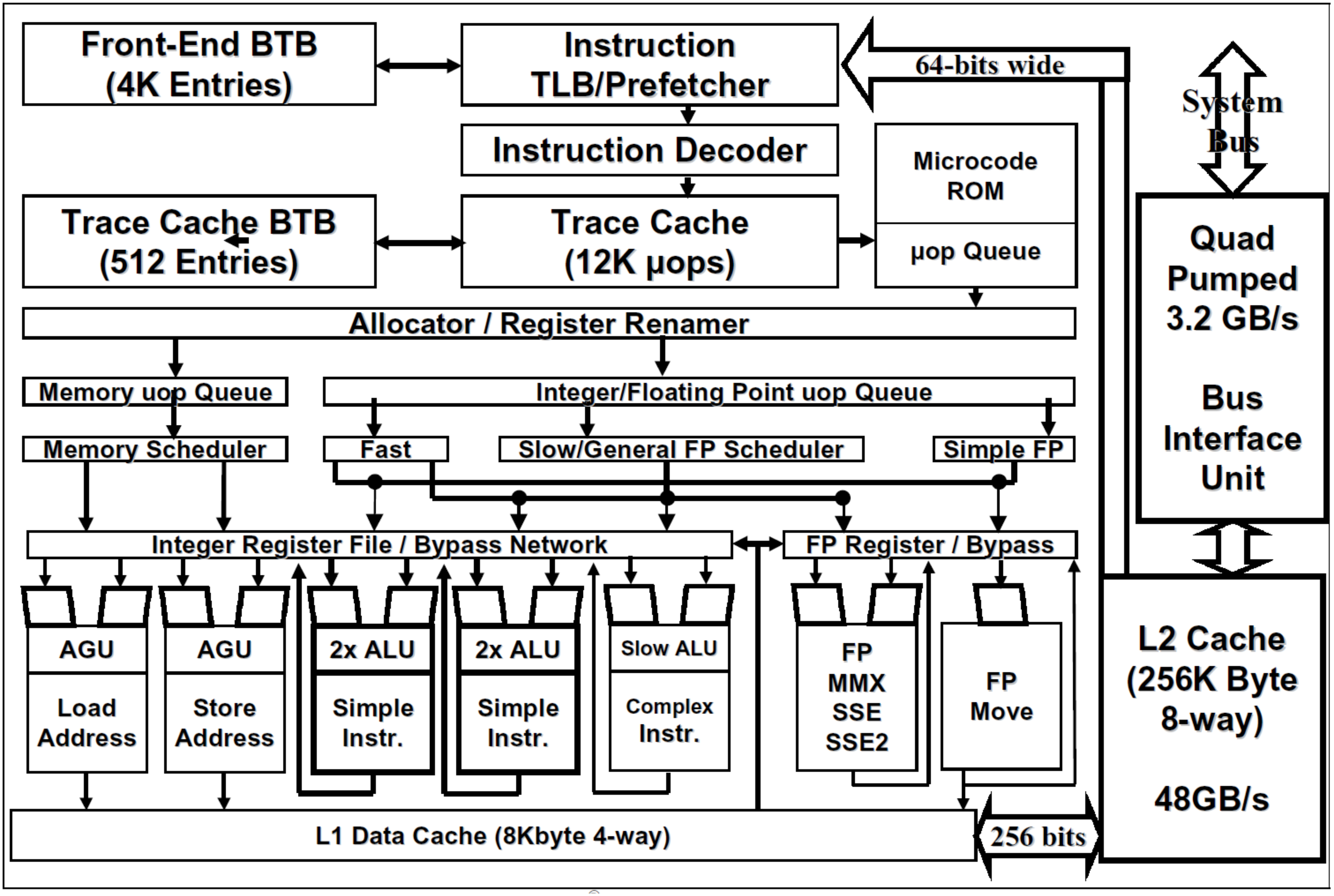
- A Register Alias Table keeps track of latest alias for logical registers
- Once retired, data is copied from the ROB to the RRF

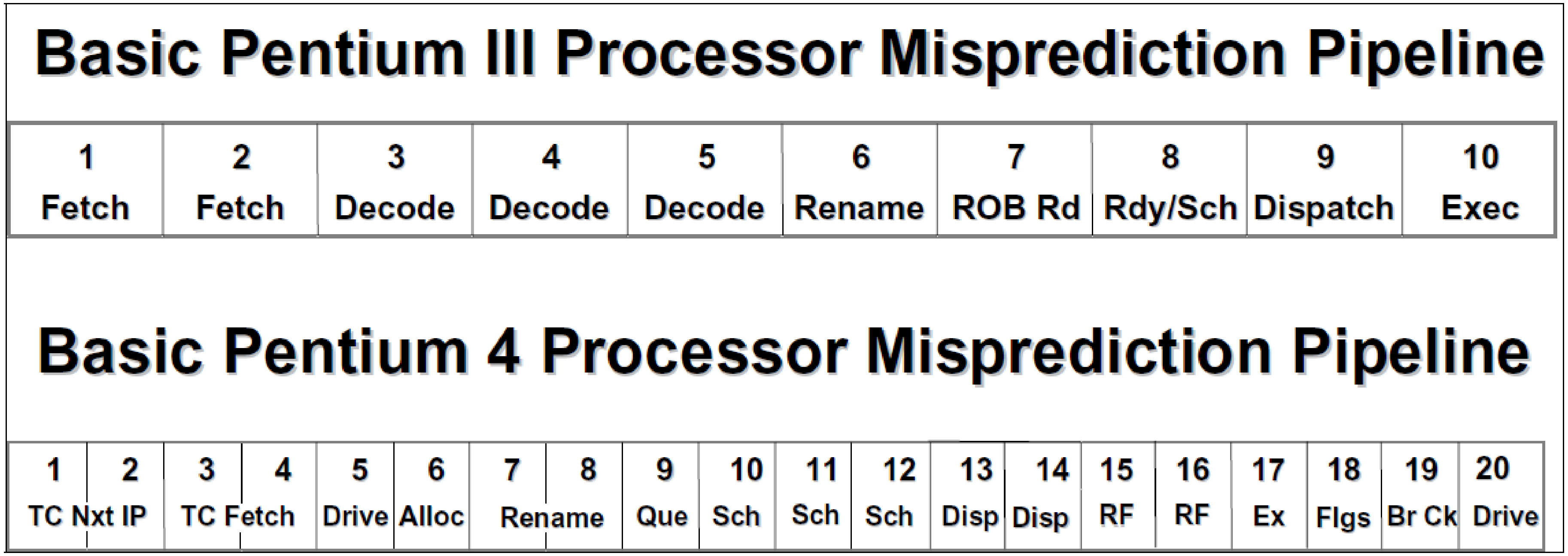
Q: How are registers allocated and freed?

- 128 Register File (RF) is separated from the ROB - which now only consists of status fields
- A unique, in-order sequence number is allocated for each uop that points to the corresponding ROB entry

See also Hsien Hsin Lee, GATech, https://slideplayer.com/slide/3388048/. Credit also to Krishna Palem

Pentium 4 (“Netburst”) microarchitecture





Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

Out-of-order processing – Four instructions per cycle

Example: Naive implementation (roughly from `cc -S`):

```
void f() {
   int i, a;
   for (i=1;
   i<=1000000000;
                    i++)
        a = a+i;
}
```

X86 code (slightly tidied but without register allocation)

```
    movl $1,-4(%ebp)
    jmp .L4
.L5
    movl -4(%ebp),%eax
    addl %eax,-8(%ebp)
    incl -4(%ebp)
.L4:
    cmpl $1000000000,-4(%ebp)
    jle .L5
```

Real example

Unoptimised:

```
    movl $1,-4(%ebp)
    jmp .L4
.L5
    movl -4(%ebp),%eax
    addl %eax,-8(%ebp)
    incl -4(%ebp)
.L4:
    cmpl $1000000000,-4(%ebp)
    jle .L5
```

5 instructions in the loop

Execution time on 2.13GHz Intel Core2Duo: 3.87 seconds (3.87 nanoseconds/iteration, 8.24 cycles)

Optimised:

```
    movl $1,%edx
.L6:
    addl %edx,%eax
    incl %edx
    cmpl $1000000000,%edx
    jle .L6
```

4 instructions in the loop, no references to main memory

Execution time on 2.13GHz Intel Core2Duo: 0.48 seconds (0.48 nanoseconds/iteration, 1.02 cycles)

Time per instruction fell: 0.77 nanoseconds to 0.12
Optimised code runs at four instructions per cycle

Resources

- Wikipedia (!):
  - http://en.wikipedia.org/wiki/Register_renaming
- Papers:
  - Instruction issue logic for high-performance, interruptable pipelined processors. G. S. Sohi, S. Vajapeyam. International Conference on Computer Architecture, 1987 (http://doi.acm.org/10.1145/30350.30354)
  - Towards Kilo-instruction processors. Cristal, Santana, Valero, Martinez ACM Trans. Architecture and Code Optimization (http://doi.acm.org/10.1145/1044823.1044825)
- Other simulators:
  - Simplescalar: *www.**simplescalar**.com/*
  - Gem5: http://www.gem5.org
  - Liberty: http://liberty.cs.princeton.edu/
  - SimFlex: http://parsa.epfl.ch/simflex/
  - SIMICS: http://www.windriver.com/products/simics/

Dynamic scheduling - summary

- Dynamic instruction scheduling is attractive:
  - Reduced dependence on compile-time instruction scheduling (and compiler knowledge of hardware)
  - Handles dynamic stalls due to cache misses
  - Register renaming frees architecture from constraints of the instruction set
- Comes with costs
  - Increases pipeline depth, and misprediction latency
  - Increased power consumption and area (but not by all that much if you are careful and clever)
  - Increased complexity and risk of design error
  - Hard to predict performance, hard to optimise code