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 2023

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/2324/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)

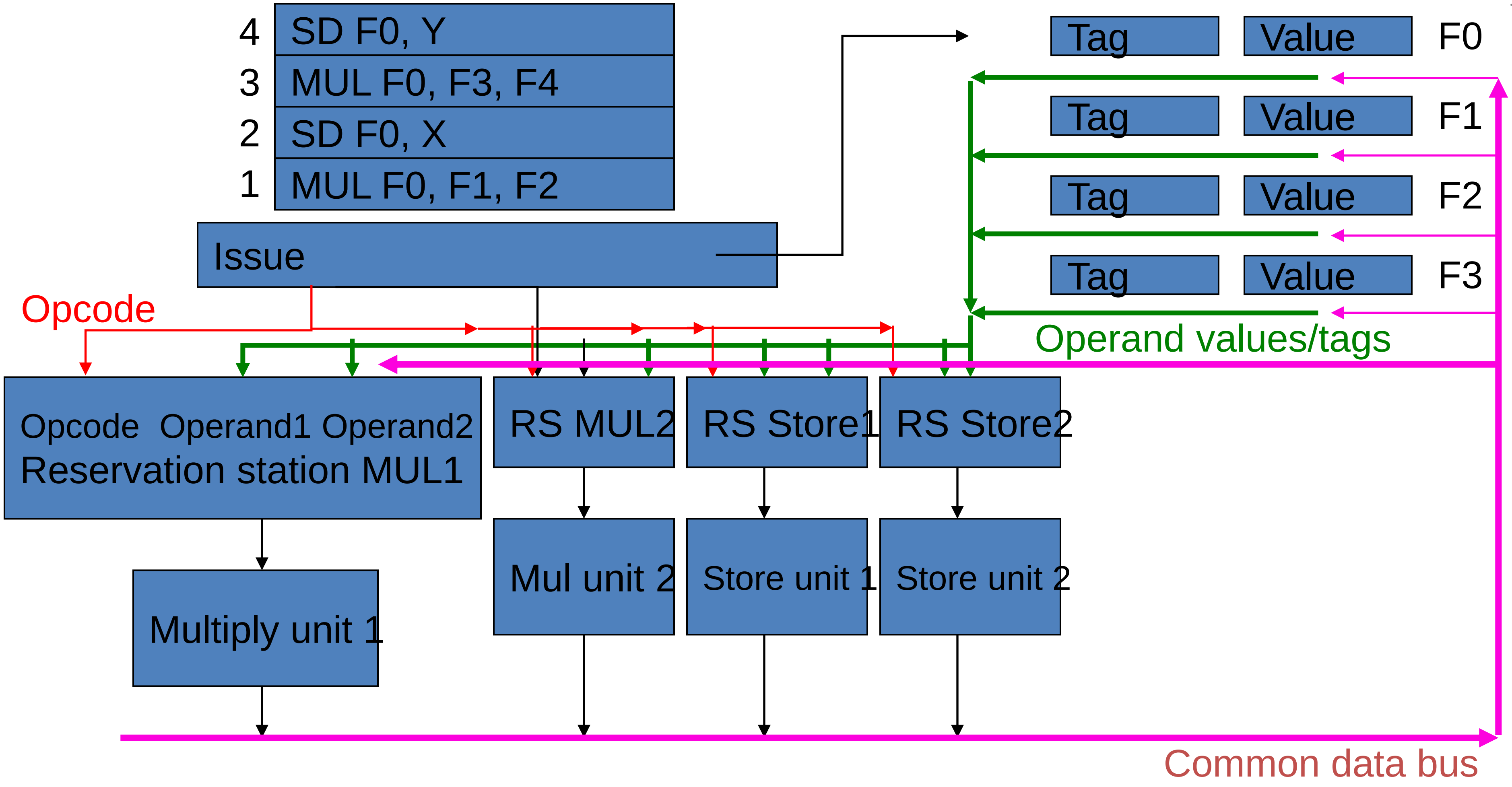
    Write on Common Data Bus to all awaiting FUs
    & reorder buffer; mark reservation station available.

4. Commit—update register with reorder result

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

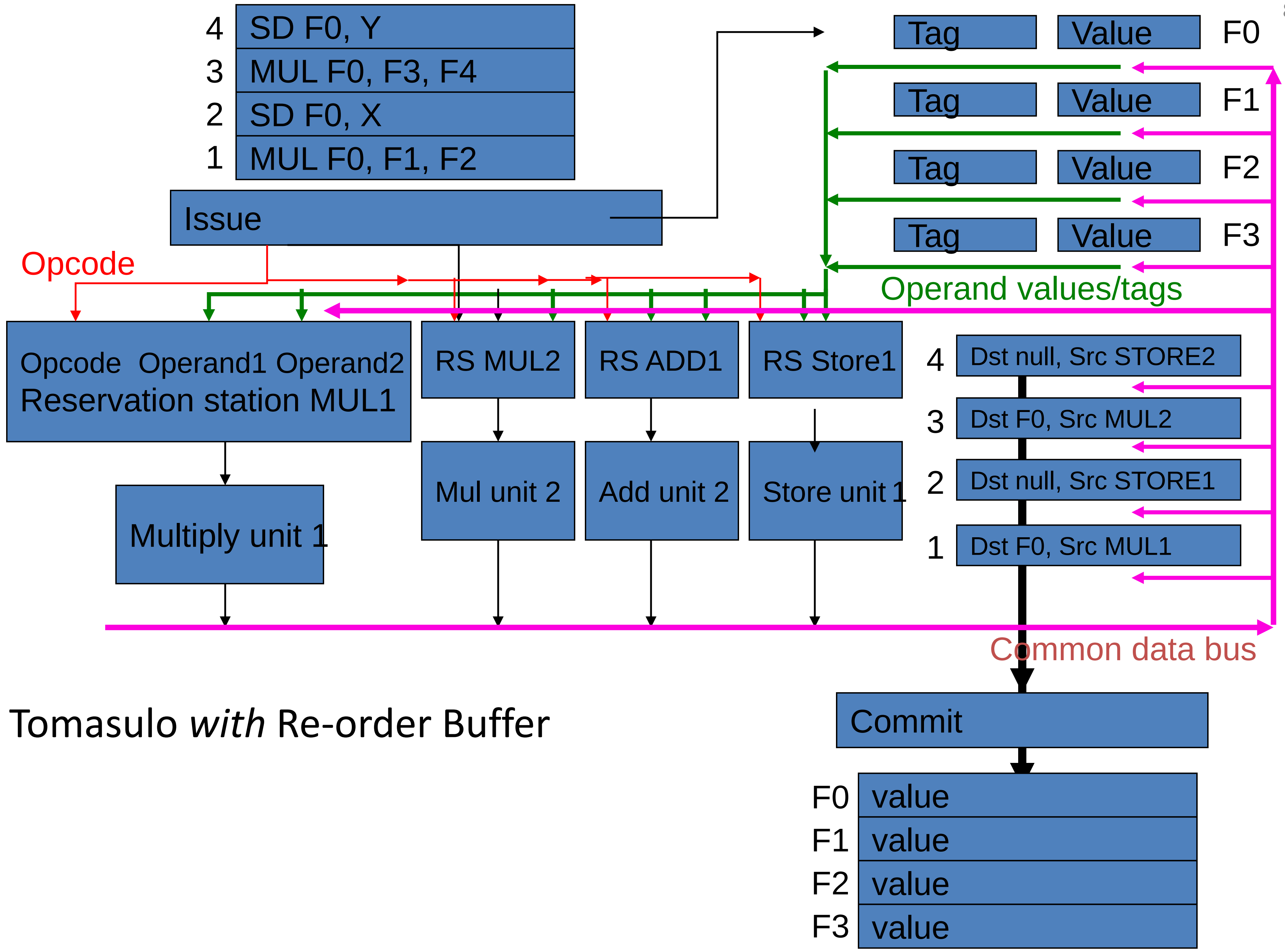
        update the (commit-side) register with the result (or store to memory), and remove the instruction from the reorder buffer.

Mispredicted branch flushes reorder buffer

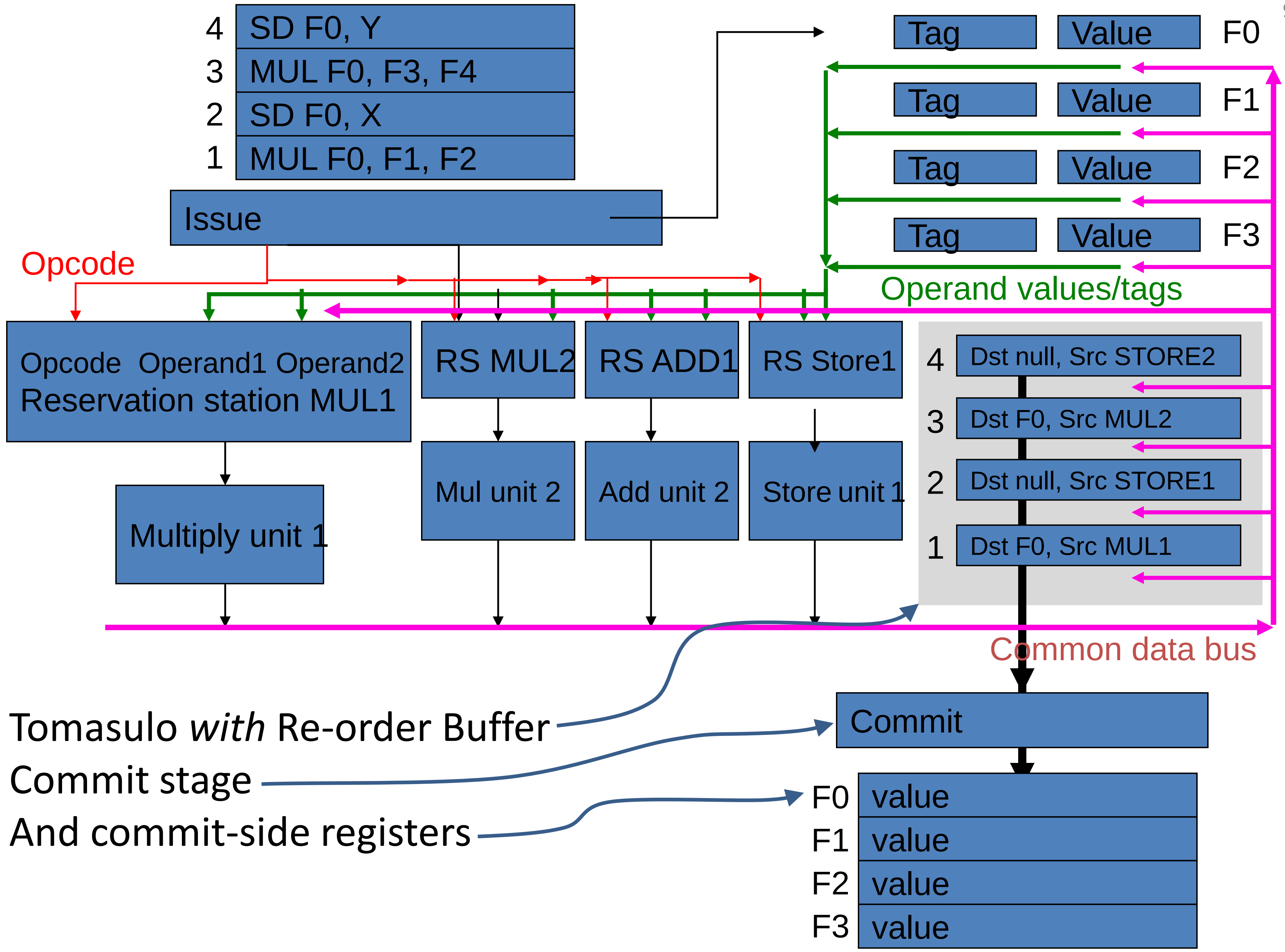


Tomasulo *without* Re-order Buffer

(from previous lecture)



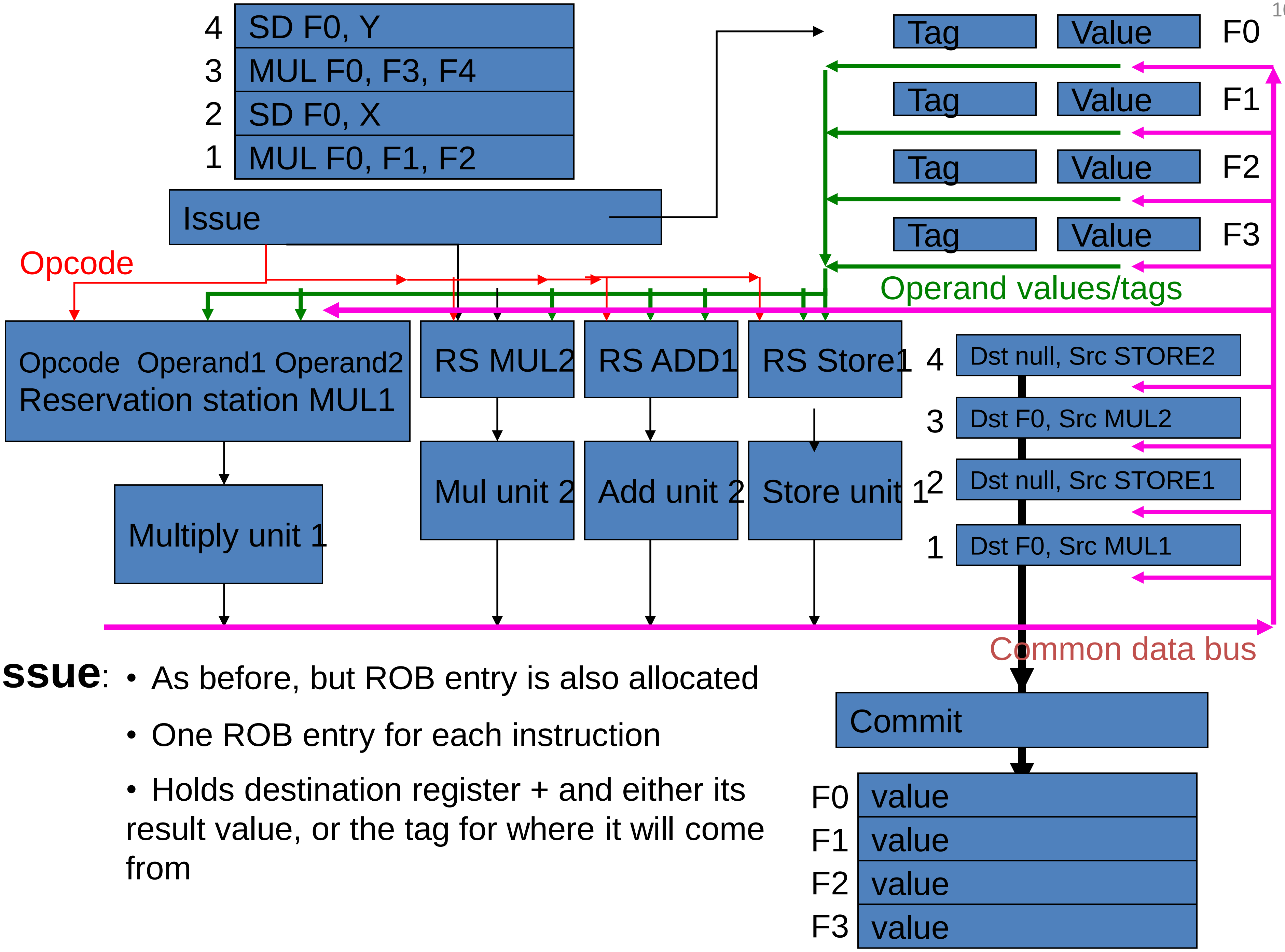
Tomasulo *with* Re-order Buffer



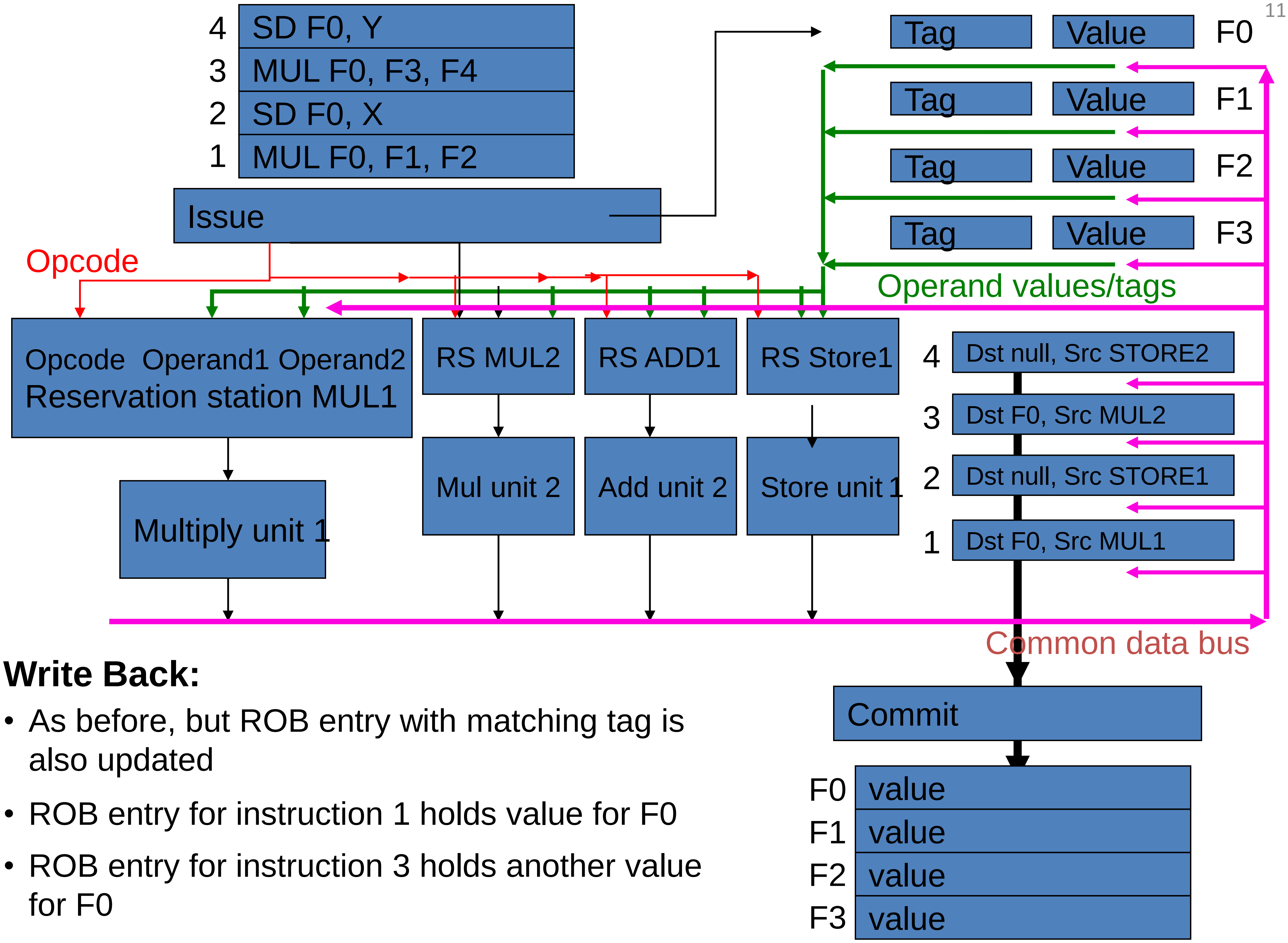
Tomasulo *with* Re-order Buffer

Commit stage

And commit-side registers

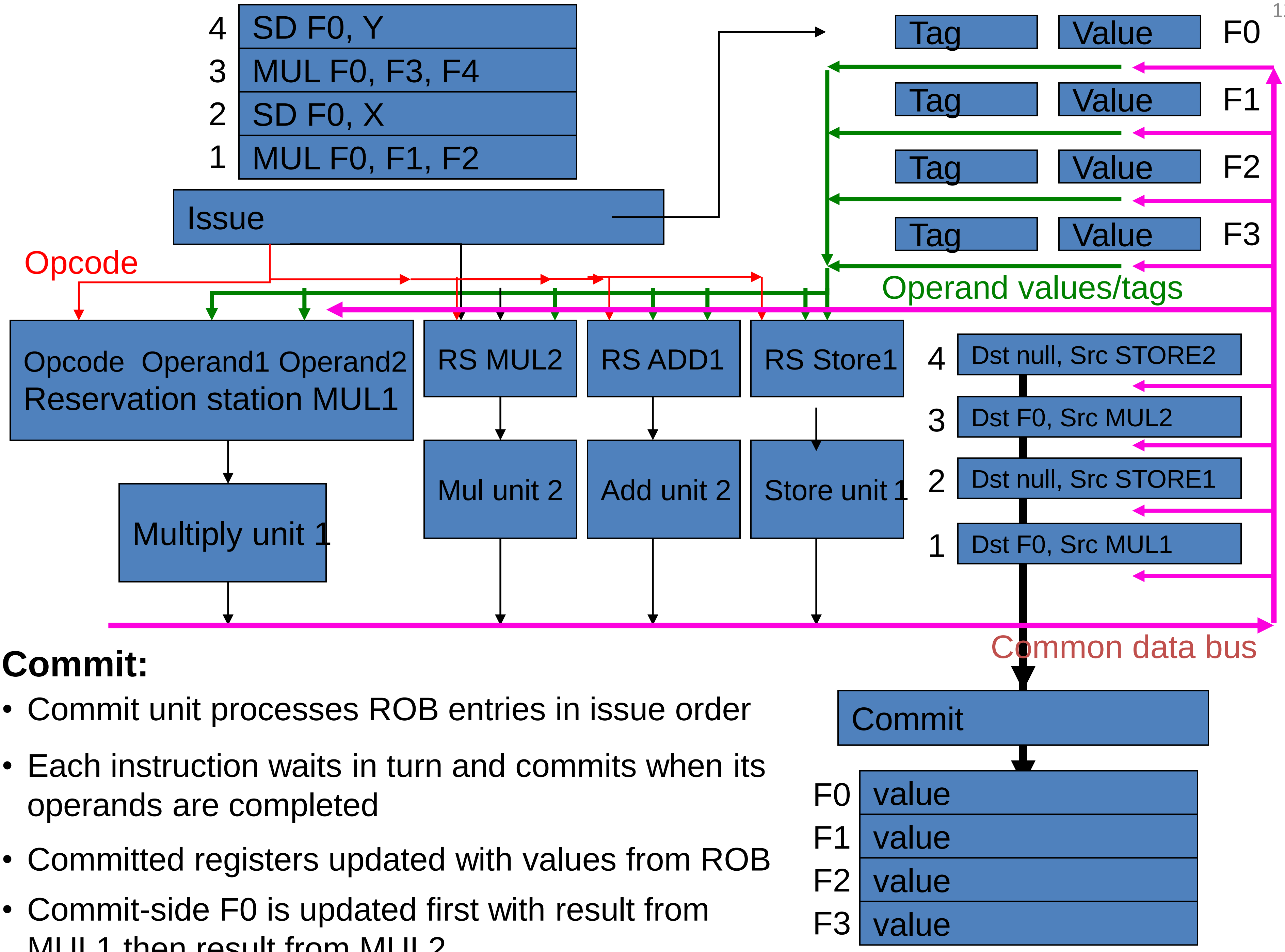


Issue:
- As before, but ROB entry is also allocated
- One ROB entry for each instruction
- Holds destination register + and either its result value, or the tag for where it will come from



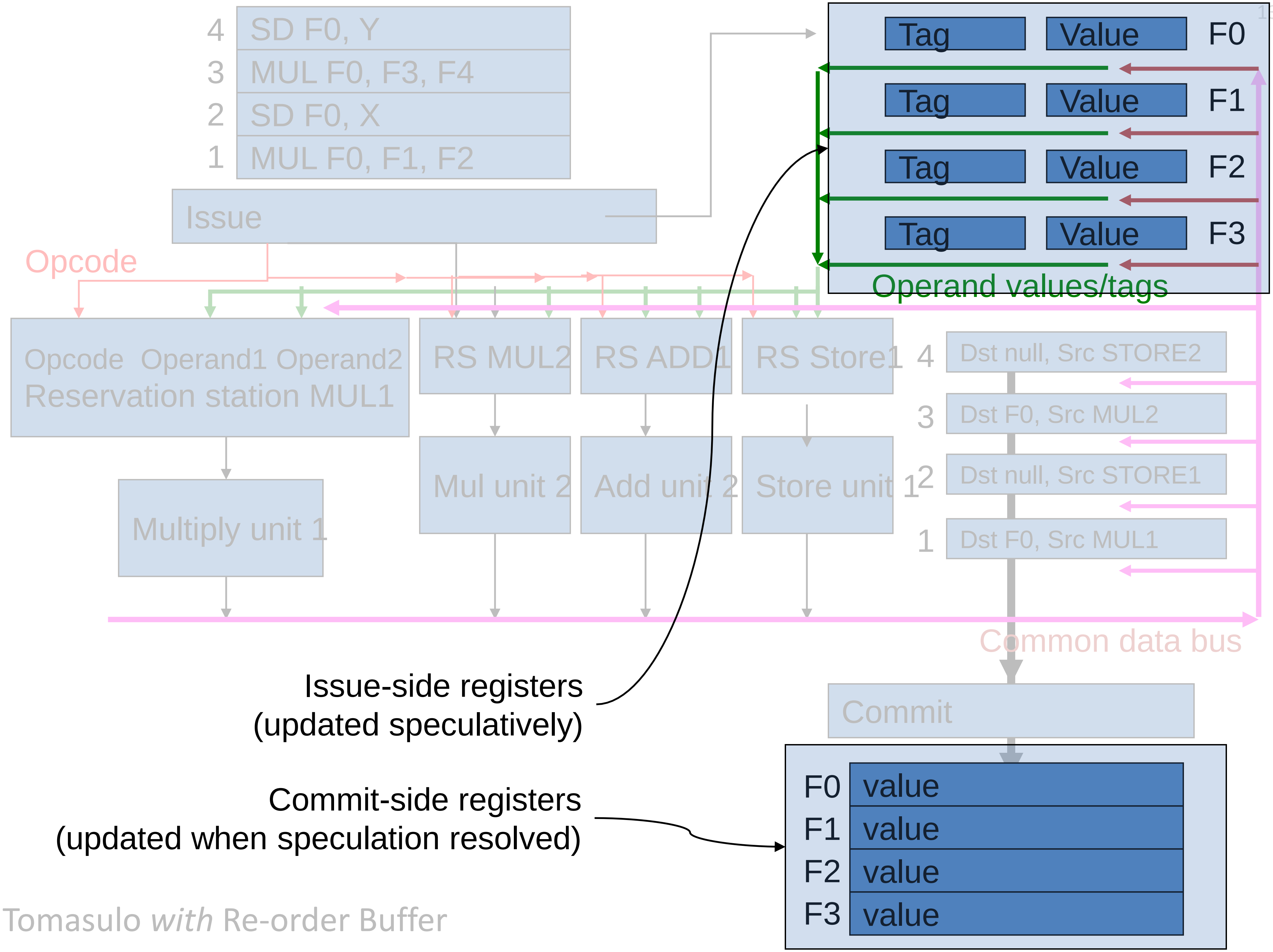
Write Back:

- 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:

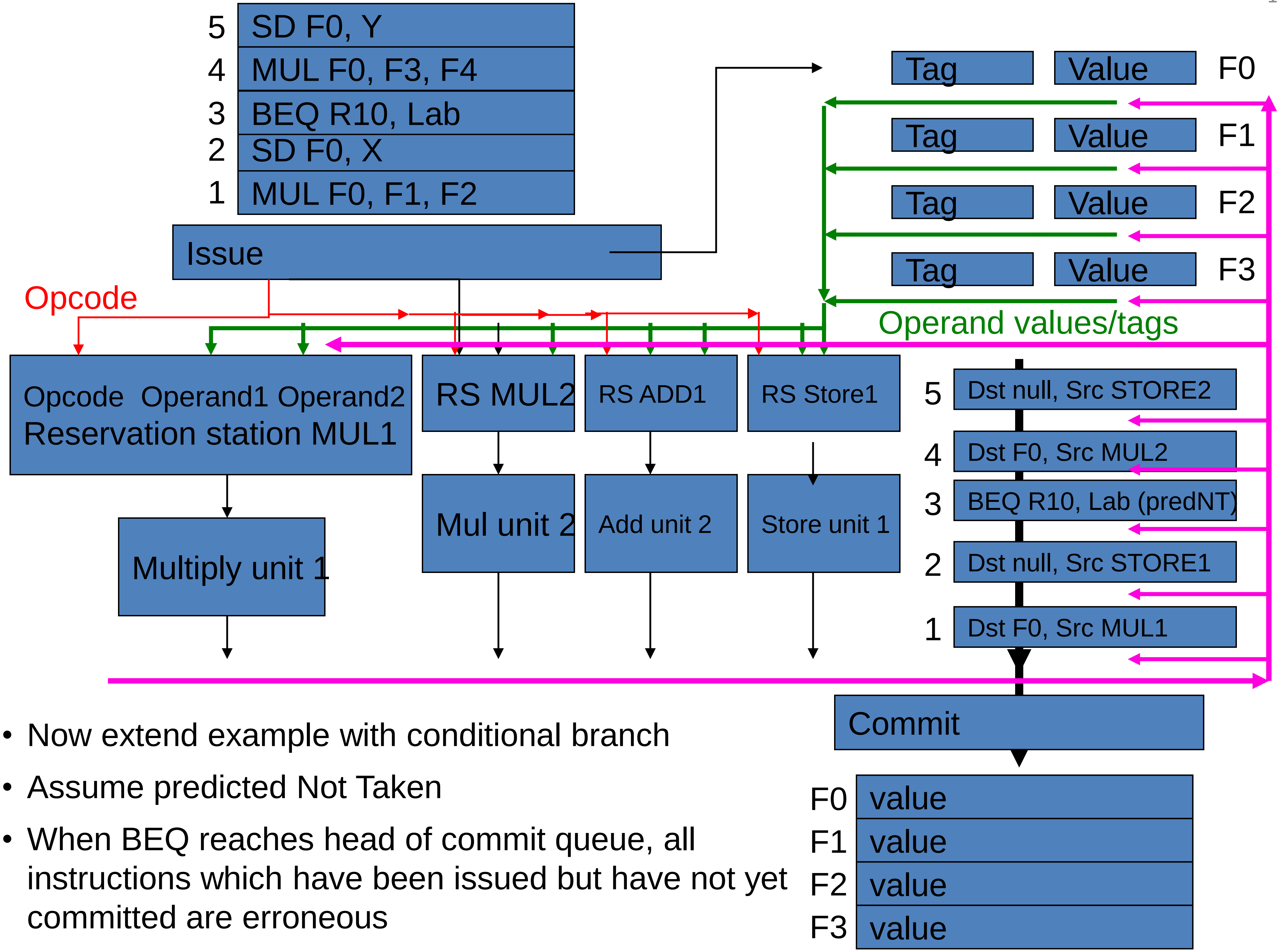
- Commit unit processes ROB entries in issue order
- Each instruction waits in turn and commits when its operands are completed
- Committed registers updated with values from ROB
- Commit-side F0 is updated first with result from MUL1 then result from MUL2



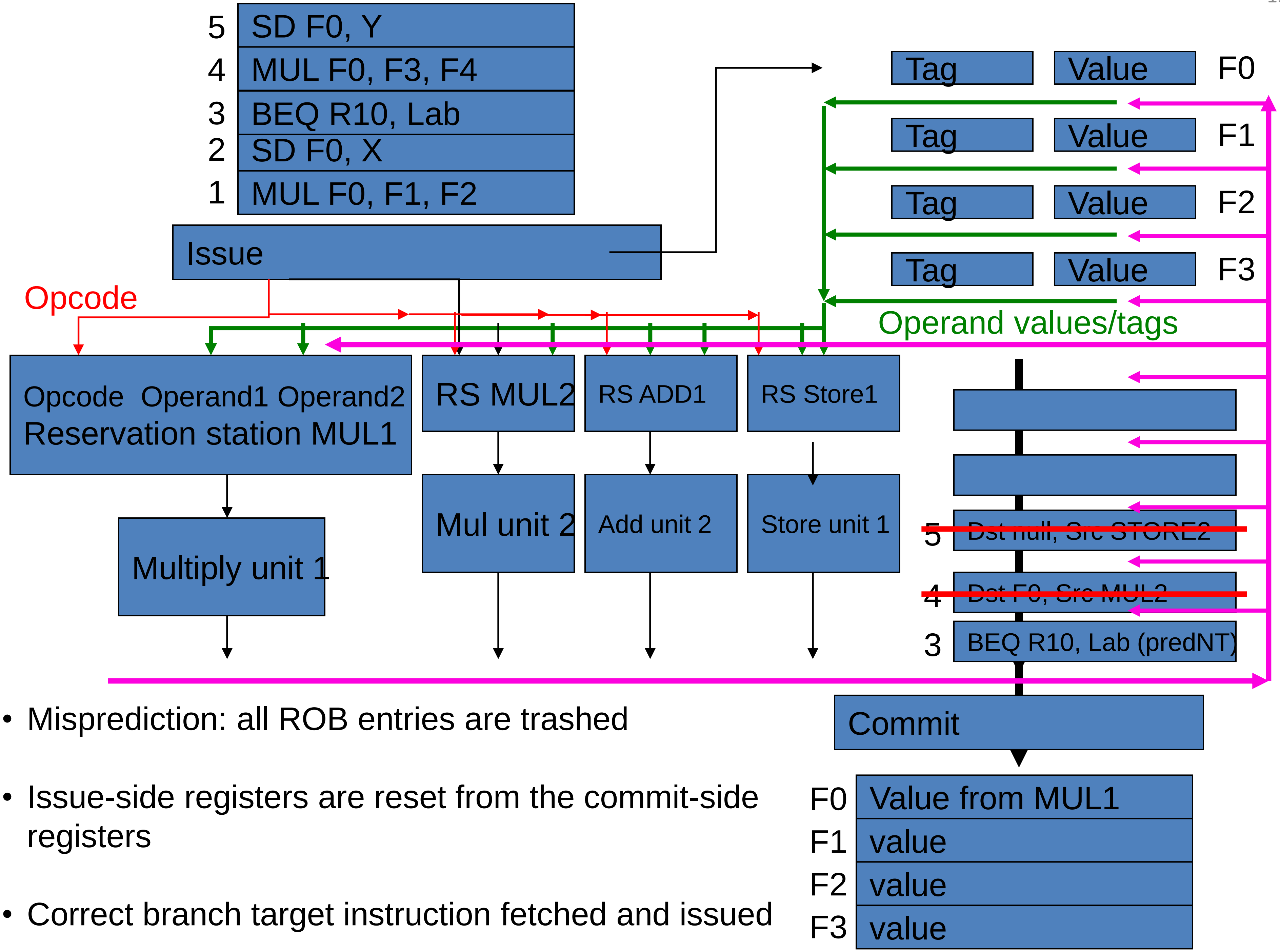
Issue-side registers
(updated speculatively)

Commit-side registers
(updated when speculation resolved)

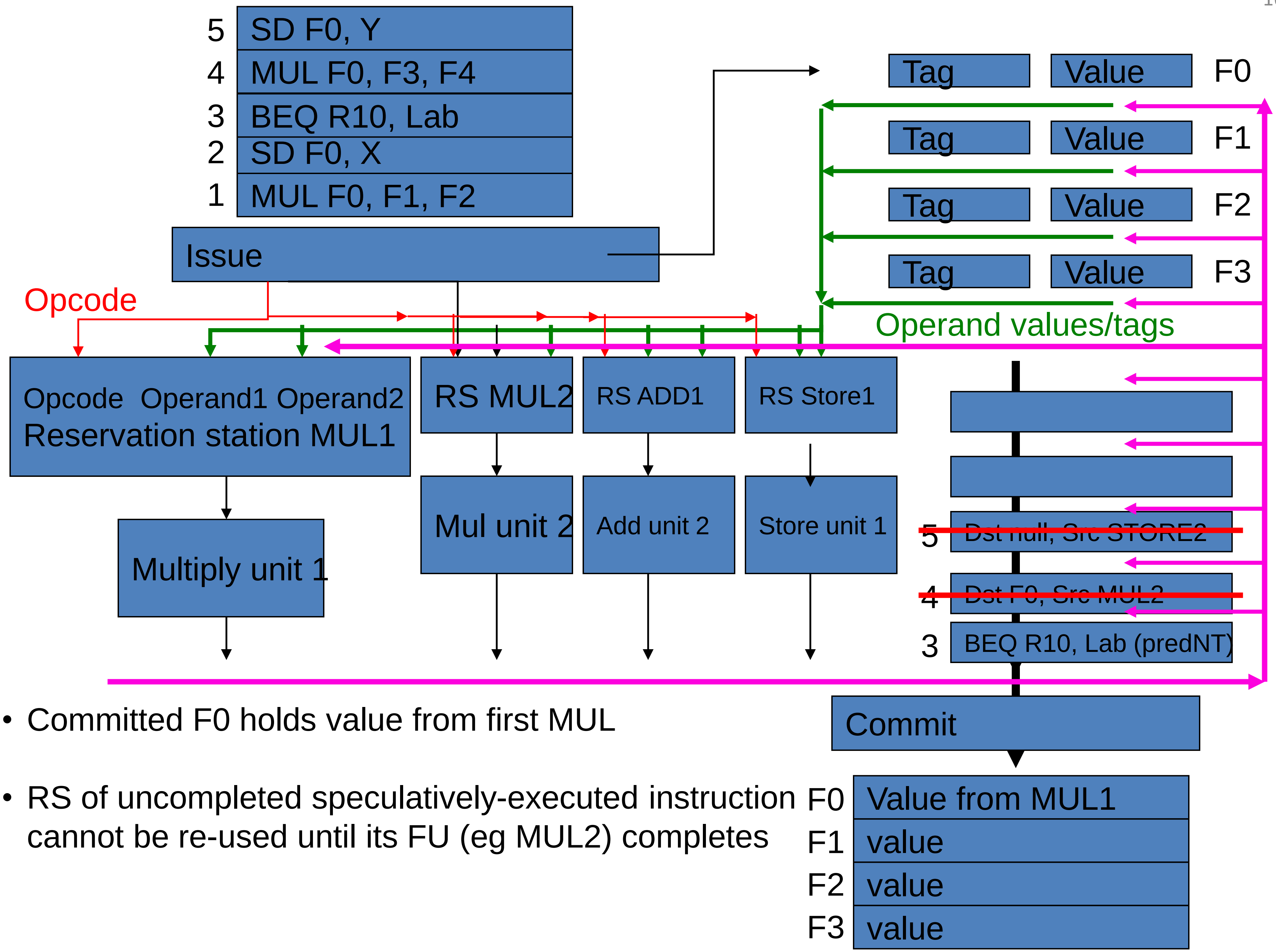
Tomasulo *with* Re-order Buffer



- 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…

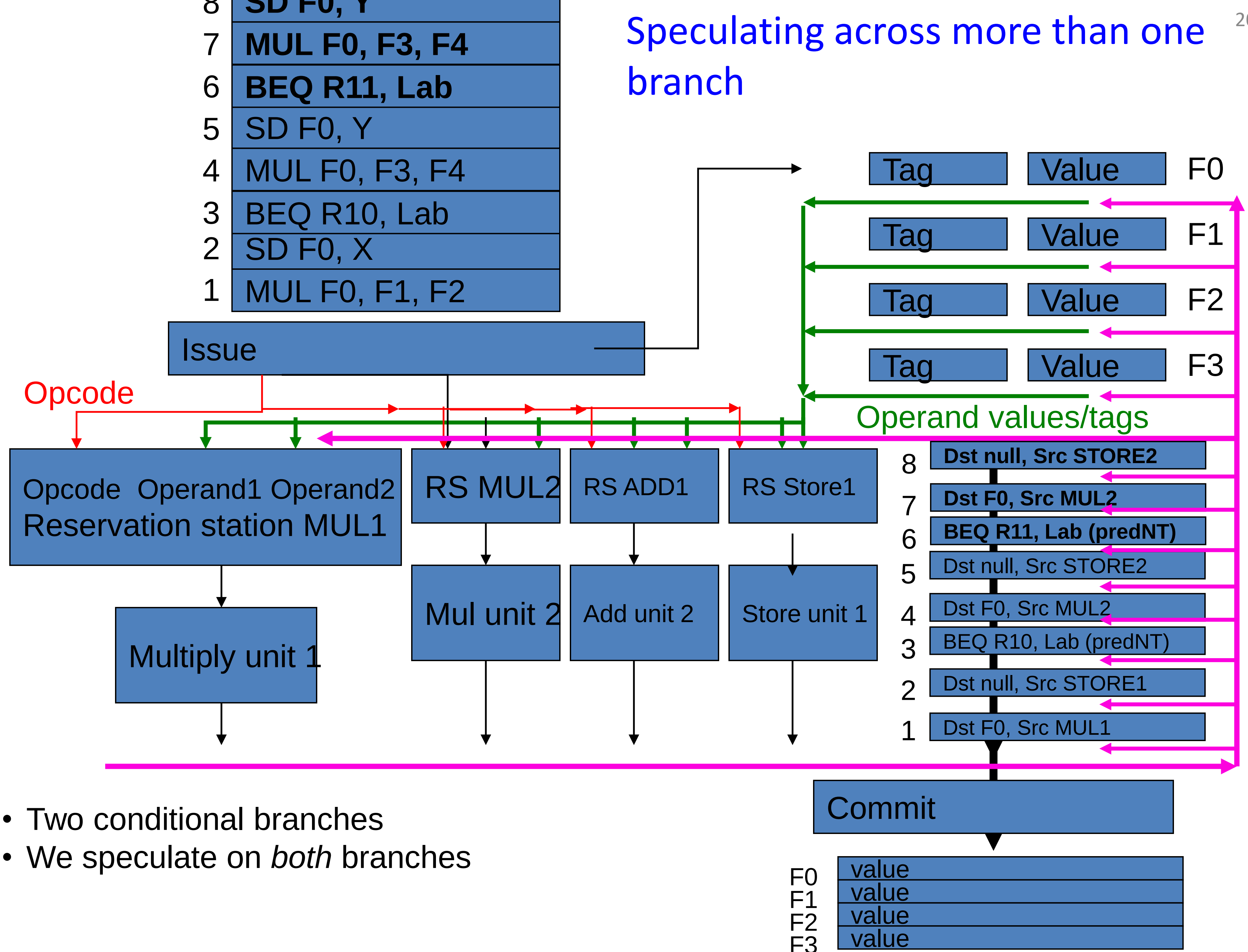
- It’s vital to reduce the branch misprediction penalty. Does the Tomasulo+ROB scheme described here roll-back as soon as the branch is found to be mispredicted?

- 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…

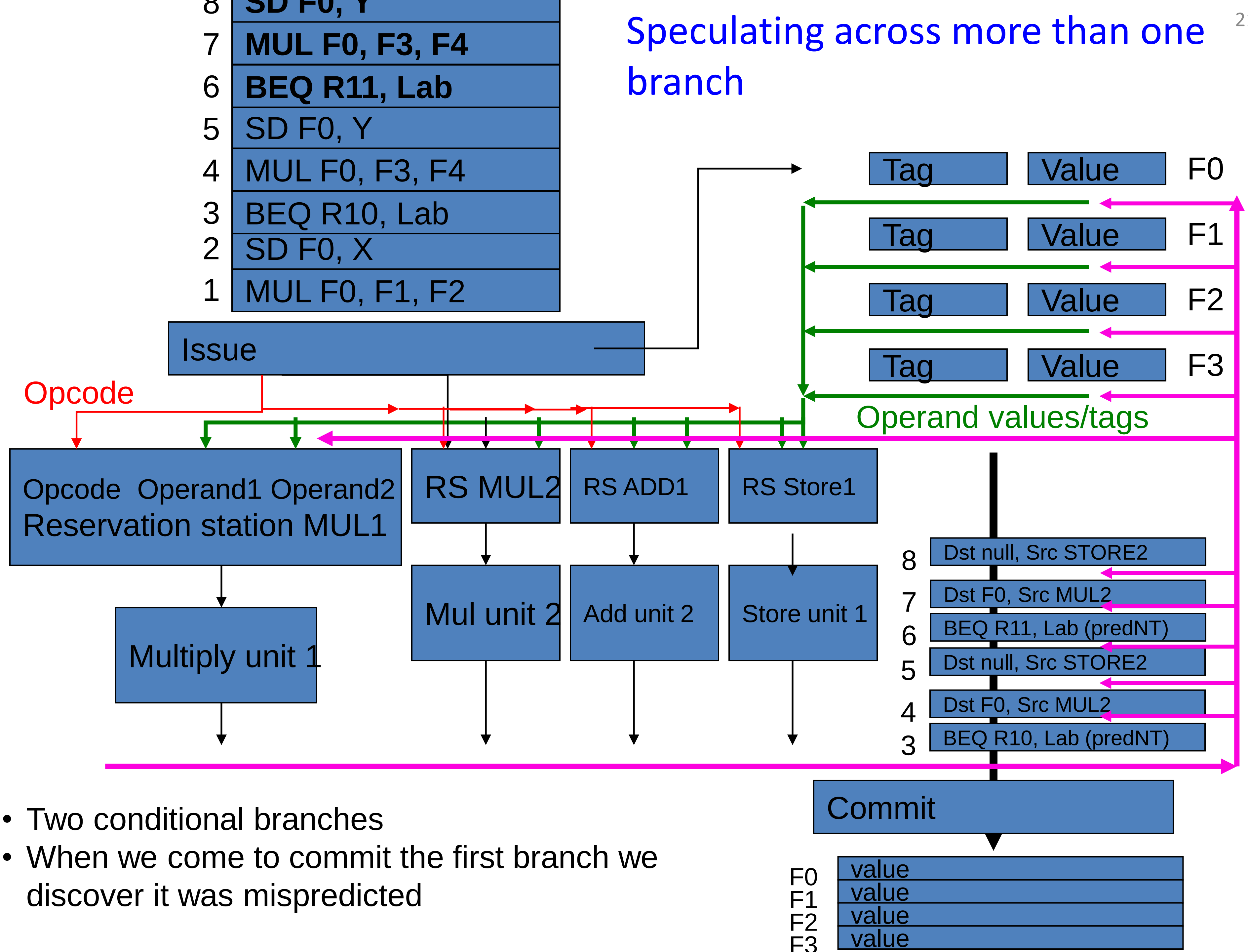
- What if a second conditional branch is encountered, before the outcome of the first is resolved?

Speculating across more than one branch



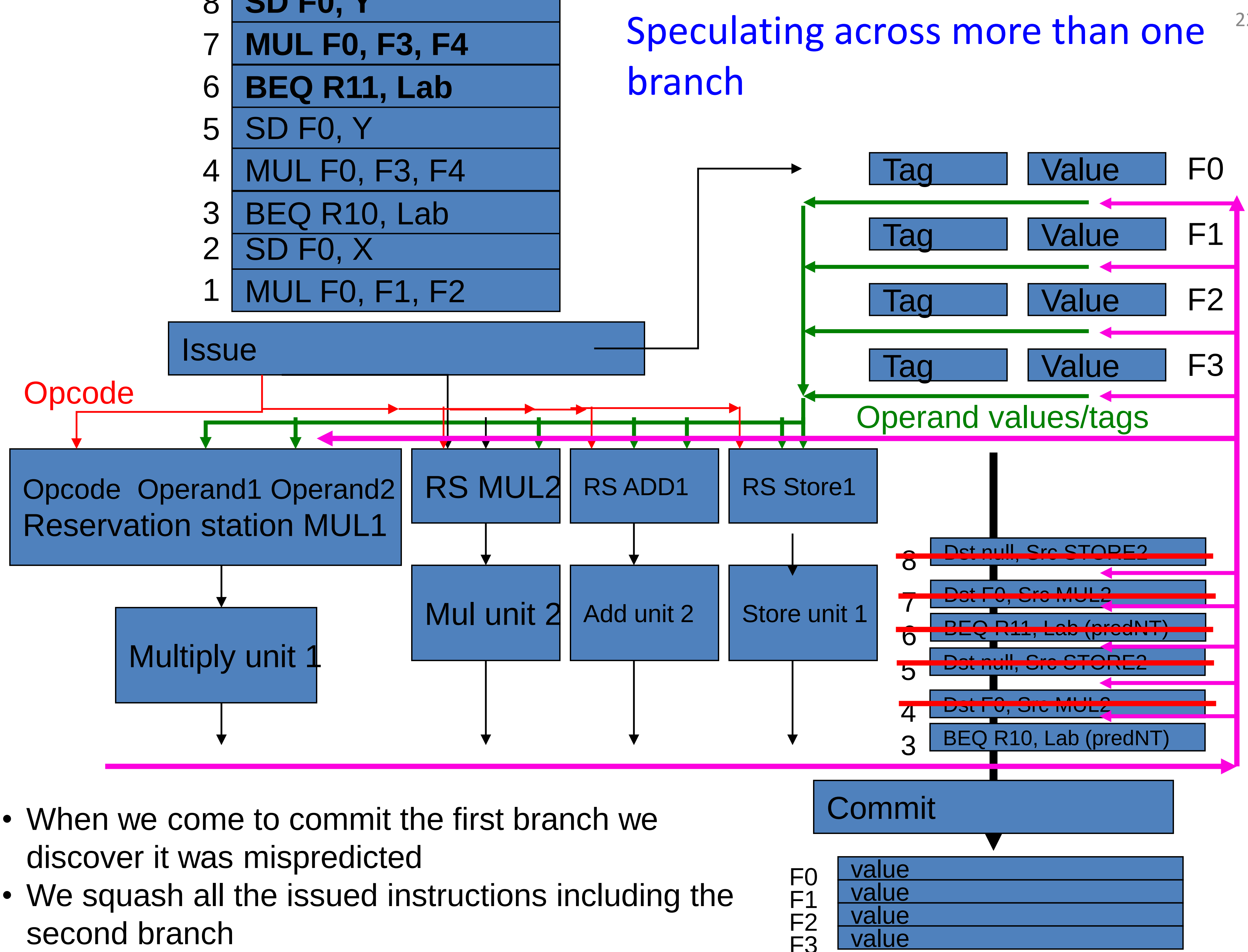
- Two conditional branches
- We speculate on *both* branches

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:

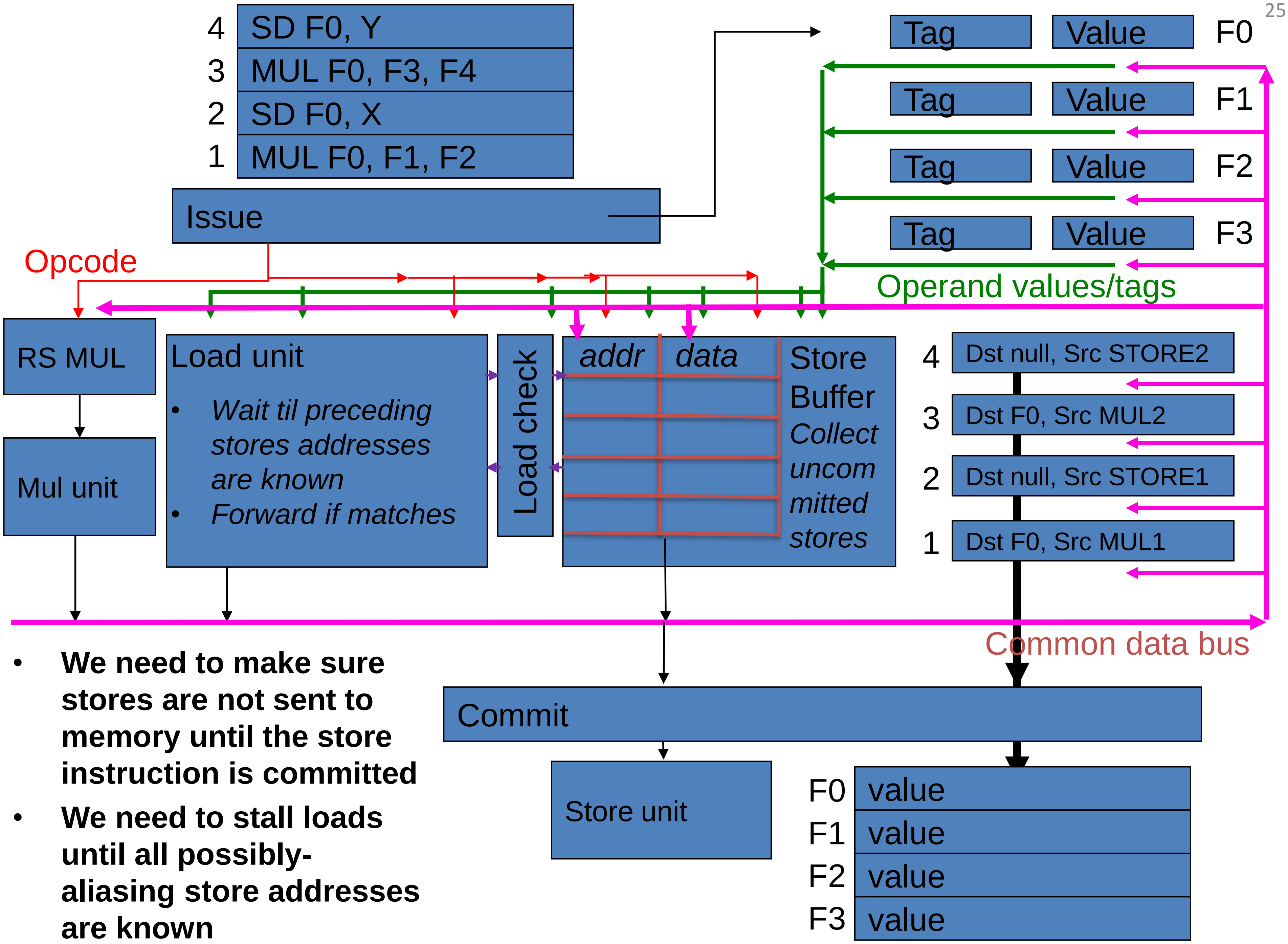
  Shen and Lipasti "Modern Processor Design" pg 271, or

  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)*

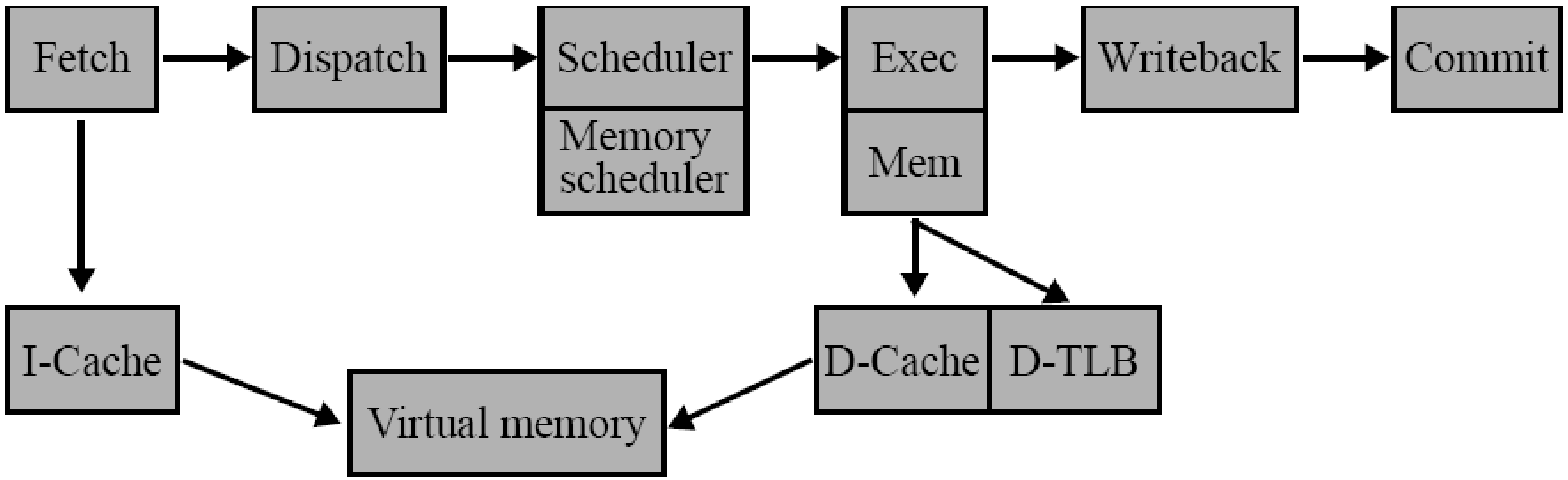
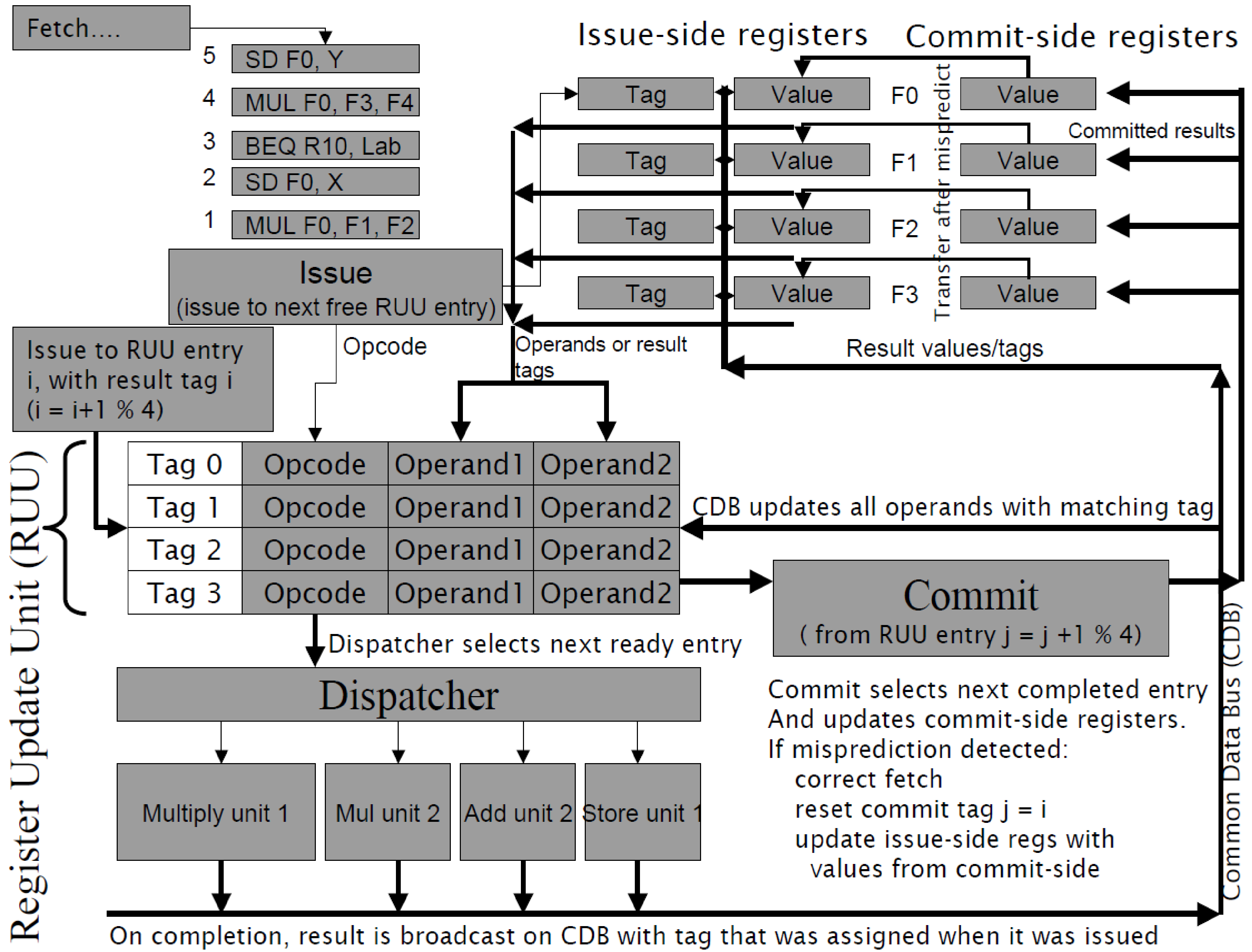


Figure 5. Pipeline for sim-outorder

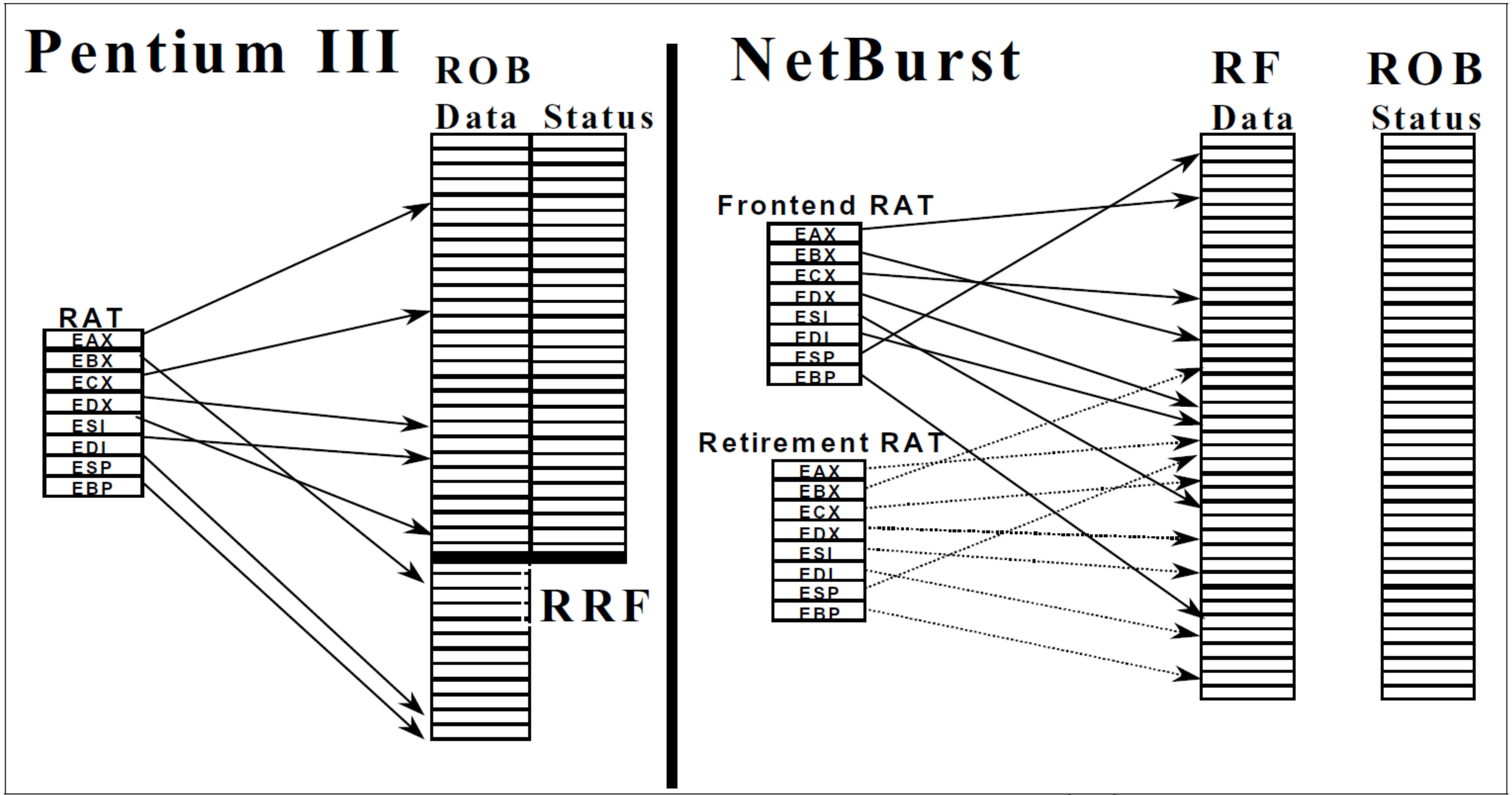
- Simplescalar is a software simulation of a processor microarchitecture
- It simulates a multi-issue out-of-order design with speculative execution
- Many aspects of the design can be controlled by parameters
- Simplescalar uses a Register Update Unit, which combines ROB and reservation stations in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



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

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
  - The ROB entry serves as a renamed register for its instruction's result
  - When an instruction completes, we still need to check whether any ROB *operands* match



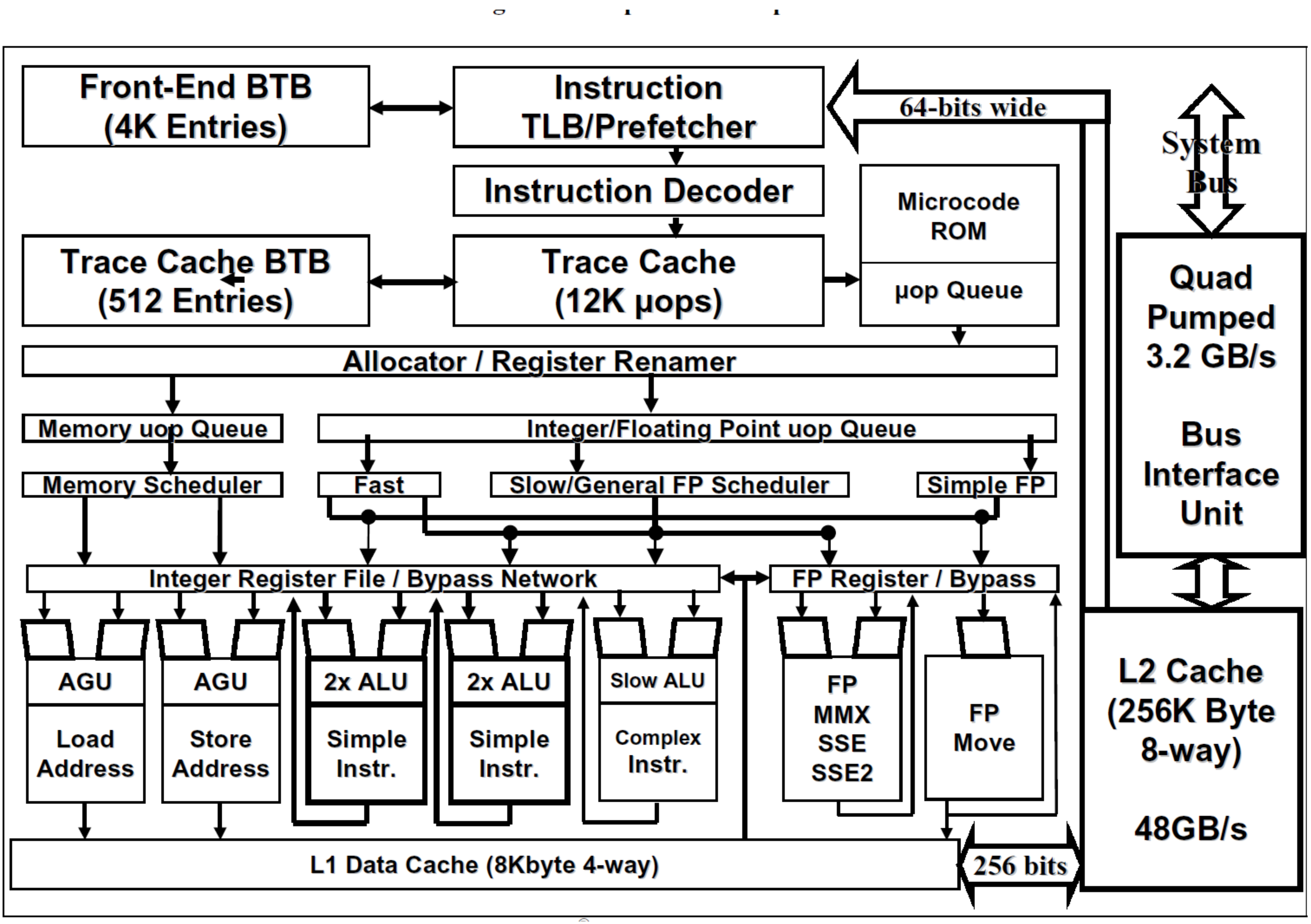
- A Register Alias Table keeps track of latest alias for logical registers
- Once retired, data is copied from the ROB to the RRF

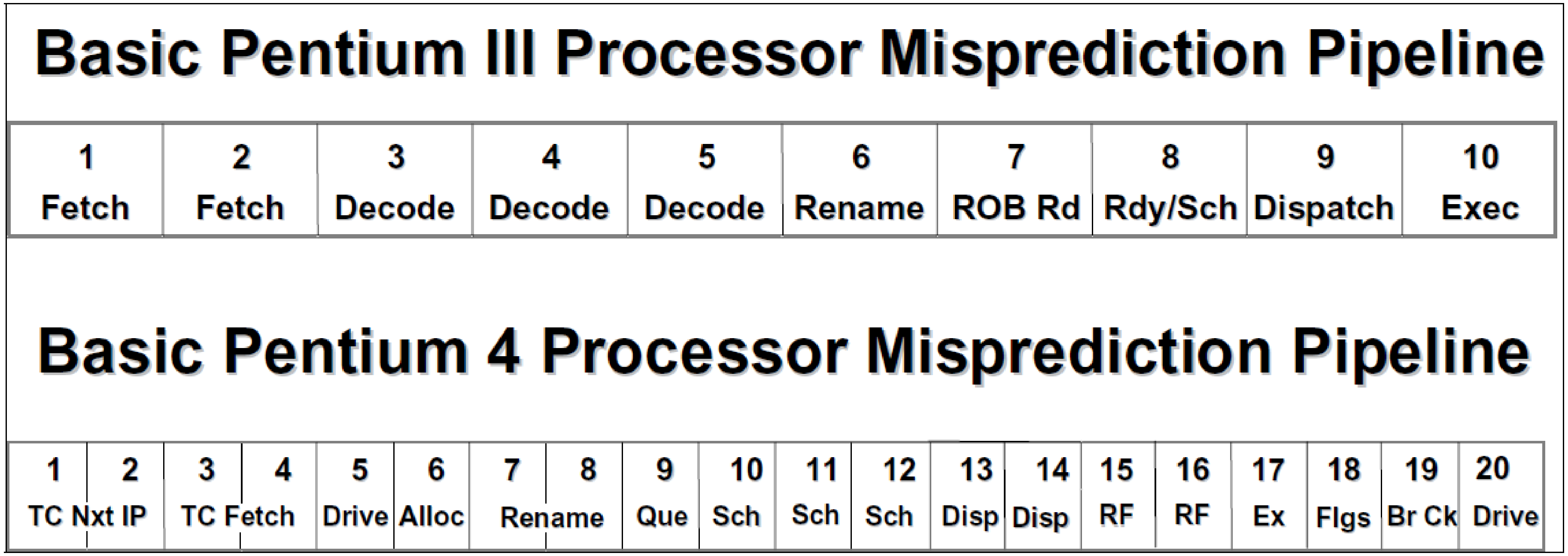
Q: How are registers allocated and freed?

- 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
- Paper:
  - The Mips R10000 superscalar microprocessor. Kenneth C. Yeager (IEEE Micro April 1996) https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp16/cse502/res/R10k.pdf
- Code:
  - https://boom-core.org/
  - https://docs.boom-core.org/en/latest/sections/reorder-buffer.html
  - https://docs.boom-core.org/en/latest/sections/reg-file-bypass-network.html
- 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

Student questions

How are interrupts the same as branches?

- An interrupt is basically a function call to an interrupt handler.It originates from some external device - perhaps a keypress or the arrival of a network packet.
- So my suggestion is that we implement it by inserting the interrupt call into the instruction stream, as if it were actually in the program.
- So the processor can execute the interrupt handler in much the same way that it would execute any function call.
- So the interrupt is introduced at issue time, in the same way that we would handle an unconditional jsr (jump subroutine, or bl on ARM).

what does this have to do with context switches?

- A context switch occurs when a thread gets an interrupt, and in the interrupt handler (ie in the OS kernel) the thread needs to block, for example because it needs to wait for an event (perhaps the arrival of a network packet).
- In this case the OS will try to find another thread to run in its place, so that the core is utilised. There might not be any other thread ready-to-run but often there is.
- So the handler will need to store all the logical registers of the old thread A to memory, and load the registers of some saved thread B's state.
- The handler can then return - this time returning into thread B.
- Some time later thread B might block in the same way, and we might re-instate thread A.
- All this register saving and restoring is done in ordinary machine code, operating on logical registers. The out-of-order execution mechanism should just do its thing and commit these instructions in the usual way.

Student question: store-to-load forwarding

The store mask records which stores a load might depend upon (line 878). I do not understand how one of the stores in the store mask can't have a valid address? Can you give me an example to understand that.

This question concerns BOOMv1, the subject of the exam in 2019-20. The relevant section of the BOOM documentation is:

This is about store-to-load forwarding. Recall that the LSU holds uncommitted stores and loads. If we have a load whose address does not match any store in the LSU, we must get its data from memory. If there is a store whose address matches, we must instead forward the data from the store to load - but we have to be sure we're forwarding from the right store, ie the youngest store whose address matches.

So what the documentation says is that when a load is decoded, we record which uncommited stores it might depend on - ie all the uncommitted older ones. The subtlety is that the addresses of the stores, and the load, might not yet be known - if they are not yet known then their "valid" bit is not set.

So you cannot safely forward from a store to a load until (1) the store's address is valid, (2) the store's data is available, (3) the load's address is valid, and (4) the addresses of all the stores that the load depends on, that are older than the matching store, are valid and don't match.

So you might wait til they are valid.

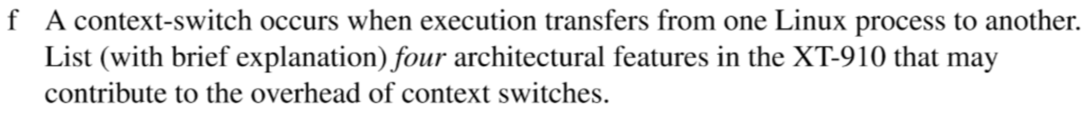
Or you might speculate before being sure!

Example:

```
R1 <- e1

R2 <- e2

R3 <- e3

ST  (R1) X

...

ST (R2) Y

...

LD R4 (R3)
```

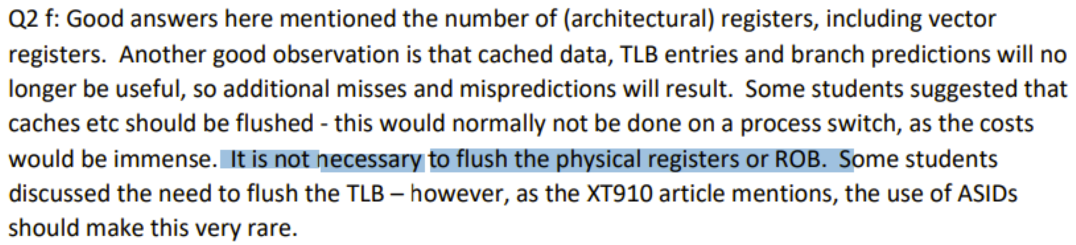
In this example, we should forward X to R4 iff e1==e3 and e2!=e3. But e1, e2, e3 might take a while to evaluated and be evaluated in an inconvenient order.

Student question: context switches



f A context-switch occurs when execution transfers from one Linux process to another. List (with brief explanation) *four* architectural features in the XT-910 that may contribute to the overhead of context switches.

For the question above, the following feedback was given:



Q2 f: Good answers here mentioned the number of (architectural) registers, including vector registers. Another good observation is that cached data, TLB entries and branch predictions will no longer be useful, so additional misses and mispredictions will result. Some students suggested that caches etc should be flushed - this would normally not be done on a process switch, as the costs would be immense. It is not necessary to flush the physical registers or ROB. Some students discussed the need to flush the TLB – however, as the XT910 article mentions, the use of ASIDs should make this very rare.

Why is it not necessary to flush the ROB on a context switch?

The intuition behind the answer given here is that what matters is the committed state of the machine. The context switch code path is "just code" so the o-o-o engine should just execute it, and commit instructions as it goes.

See for example this context switch code from Linux:
https://elixir.bootlin.com/linux/latest/source/arch/x86/entry/entry_64.S#L237

However the larger context involved in doing a context switch probably also involves the use of locks (eg on the data structures that determine which process to switch to). See:
https://elixir.bootlin.com/linux/v2.6.36/source/kernel/sched.c#L2834
Here we see, for example, the use of a "barrier()" - which, at least on some processors, might cause the ROB to be flushed (or something more subtle - like that all uncommitted memory accesses would have to be committed before continuing). Exactly what needs to be done depends on the memory consistency model supported by the processor.

This does indeed get quite complicated - if you're curious, see https://www.kernel.org/doc/Documentation/memory-barriers.txt (coauthor Will Deacon is a DoC graduate).