

COMP60001/COMP70086

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 2026
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/2627/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)

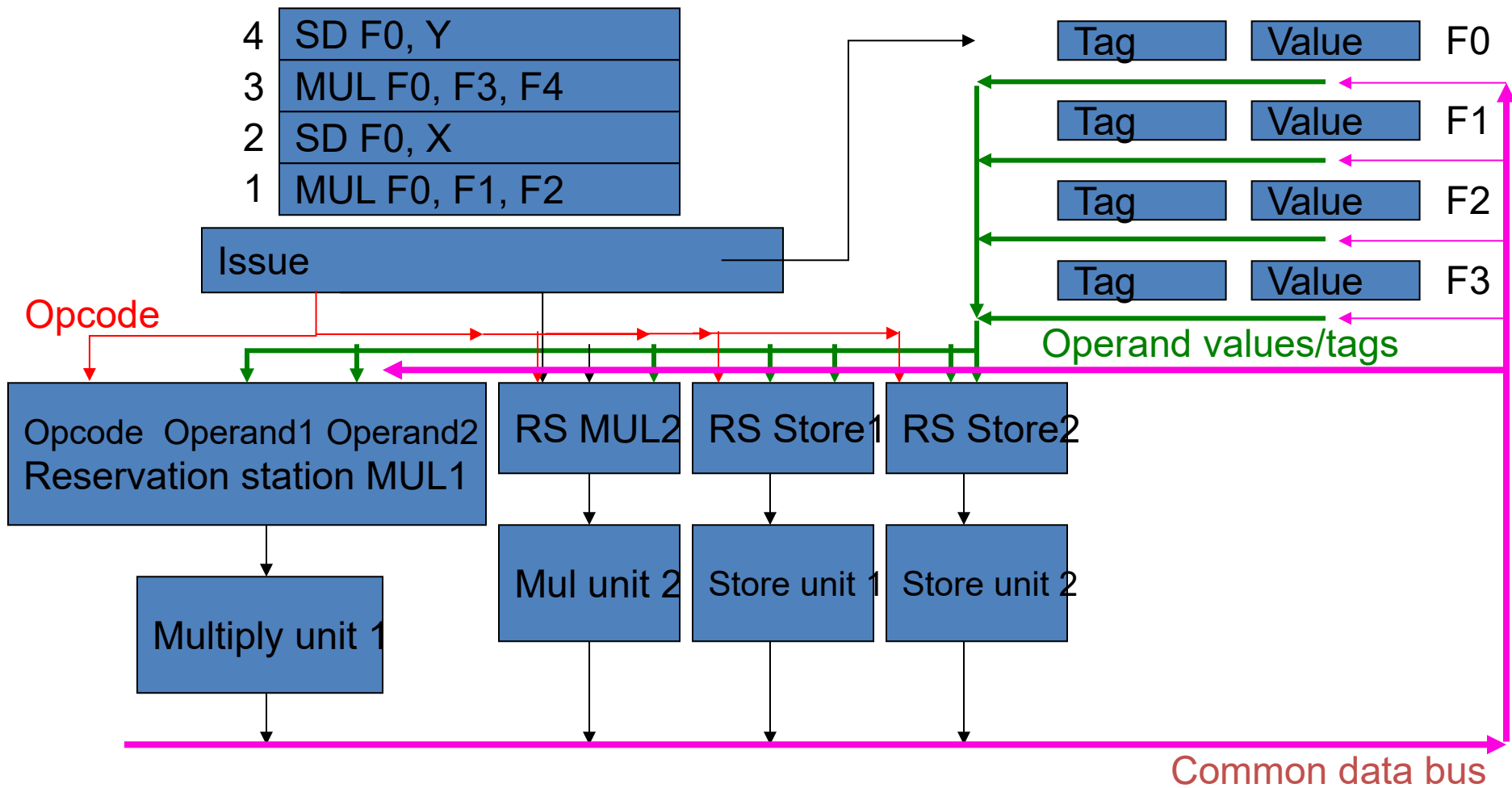
Write on Common Data Bus to all awaiting FUs **& reorder buffer**; mark reservation station available.

4. Commit—**update register with reorder result**

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

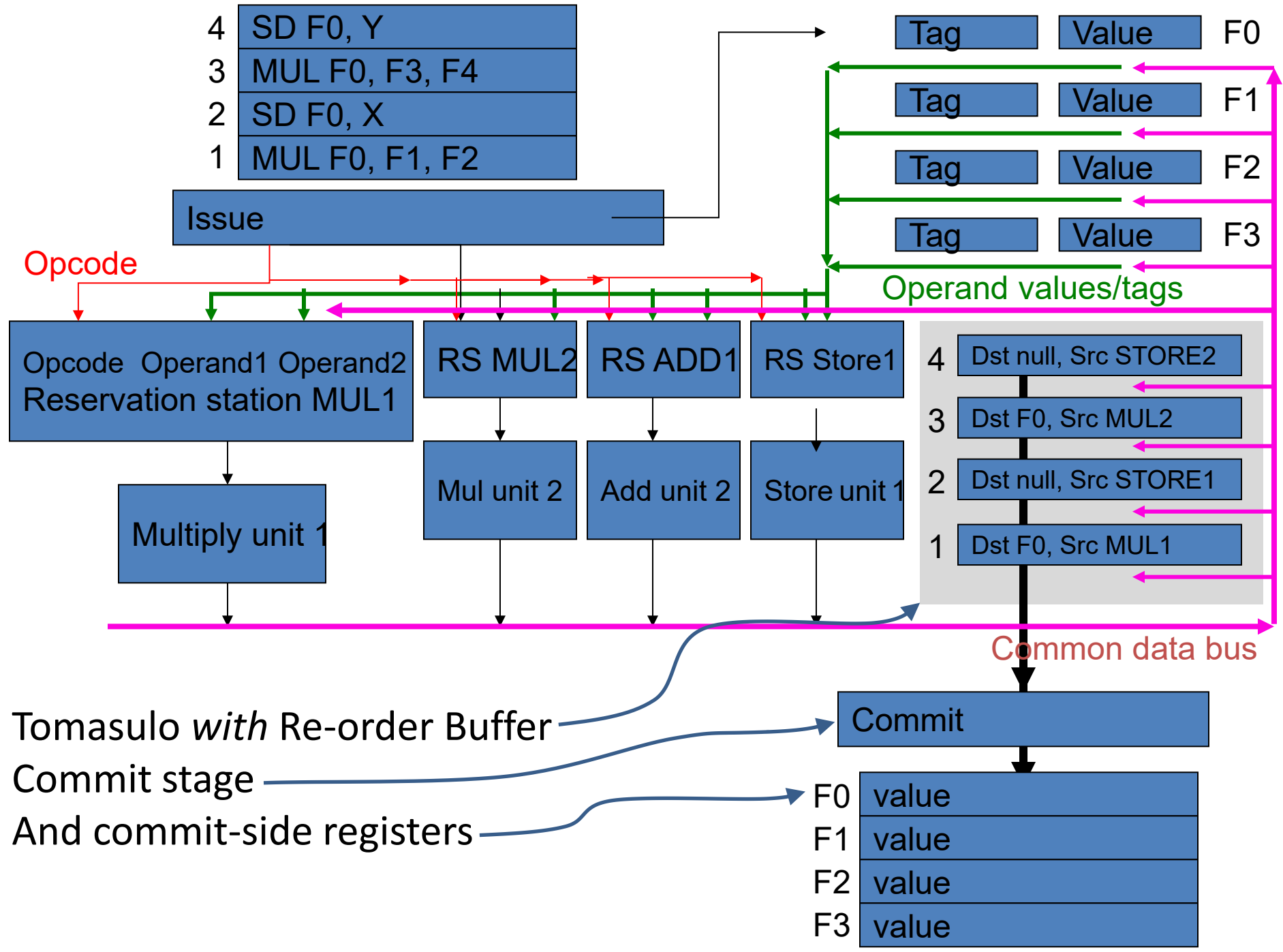
update the (commit-side) register with the result (or store to memory), and remove the instruction from the reorder buffer.

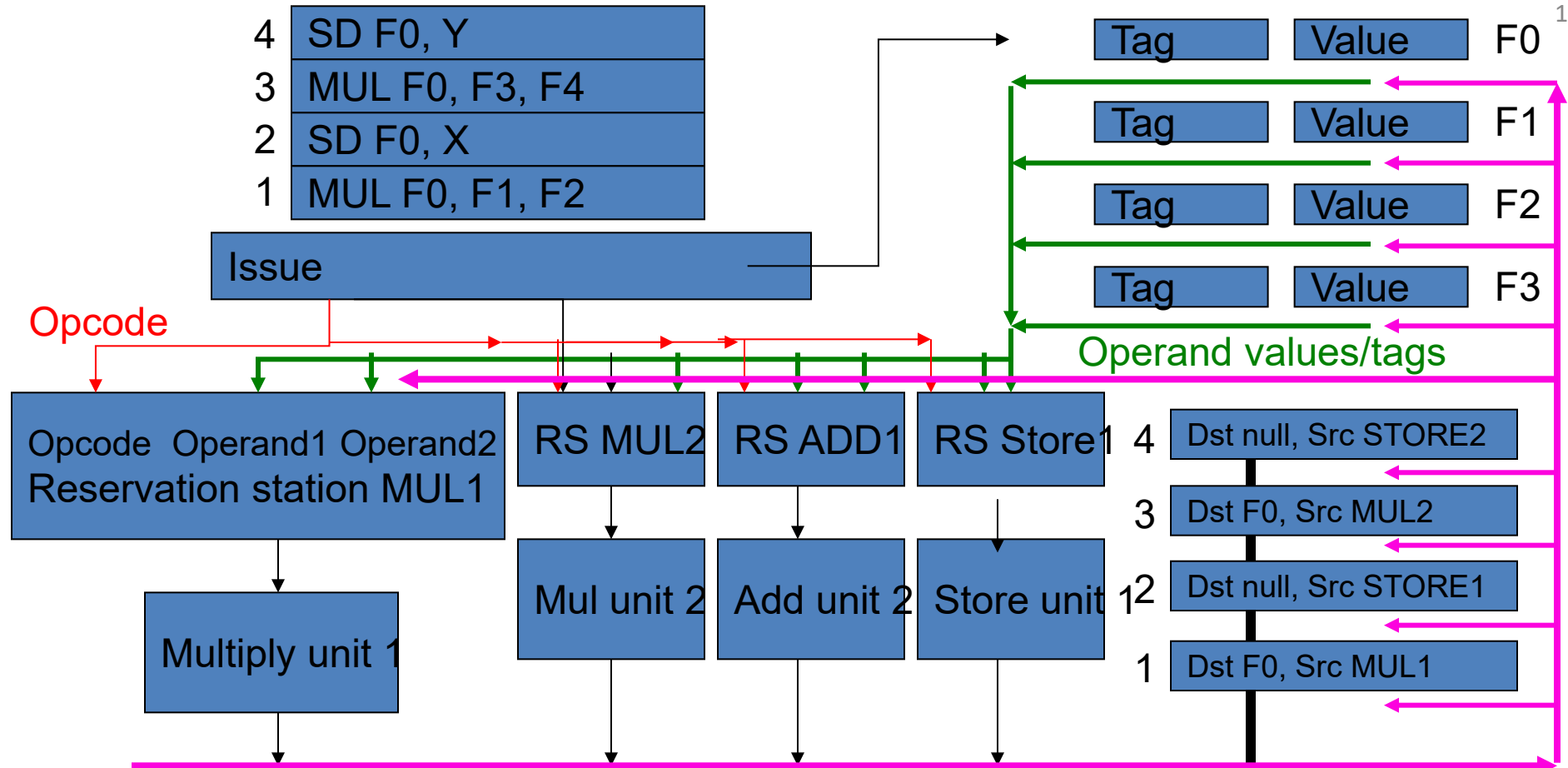
Mispredicted branch flushes reorder buffer



Tomasulo *without* Re-order Buffer

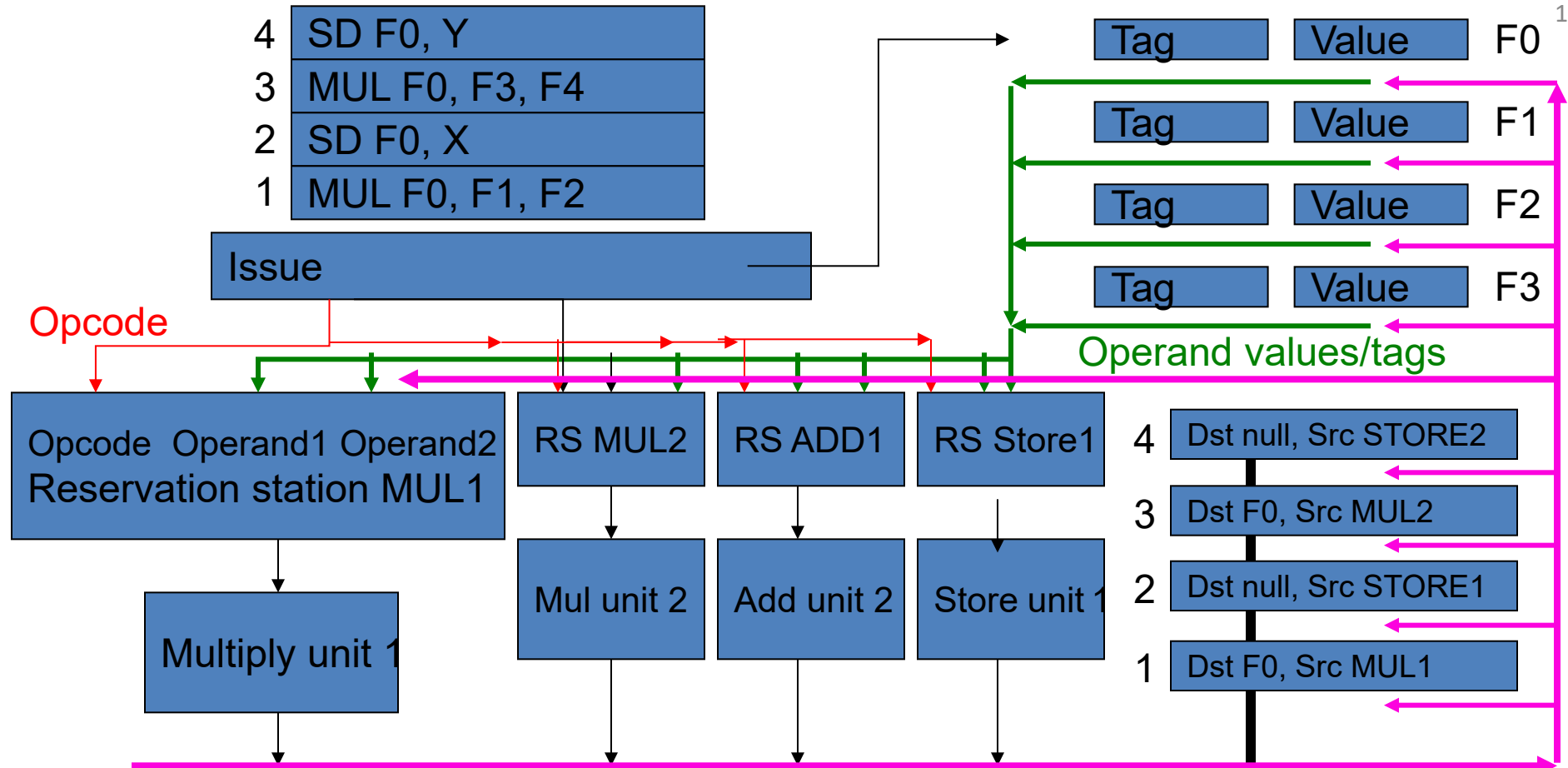
(from previous
lecture)





- 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

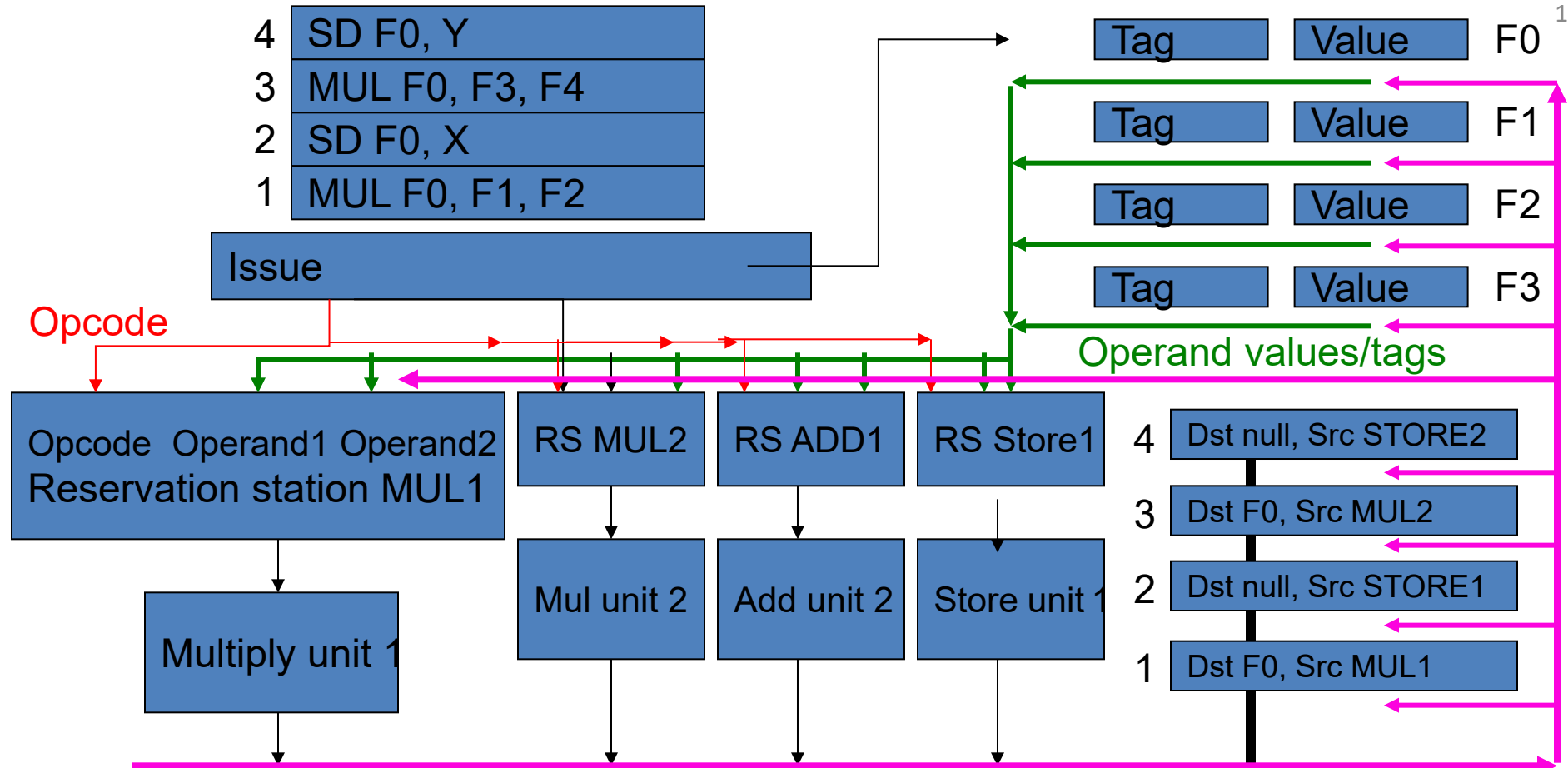
F0	value
F1	value
F2	value
F3	value



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	
F0	value
F1	value
F2	value
F3	value



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

Commit	
F0	value
F1	value
F2	value
F3	value

4	SD F0, Y
3	MUL F0, F3, F4
2	SD F0, X
1	MUL F0, F1, F2

Issue

Tag	Value	F0
Tag	Value	F1
Tag	Value	F2
Tag	Value	F3

Operand values/tags

Opcode

Opcode	Operand1	Operand2
Reservation station	MUL1	

RS	MUL2
----	------

RS	ADD1
----	------

RS	Store1
----	--------

4	Dst null, Src STORE2
3	Dst F0, Src MUL2
12	Dst null, Src STORE1
1	Dst F0, Src MUL1

Multiply unit 1

Mul unit 2

Add unit 2

Store unit

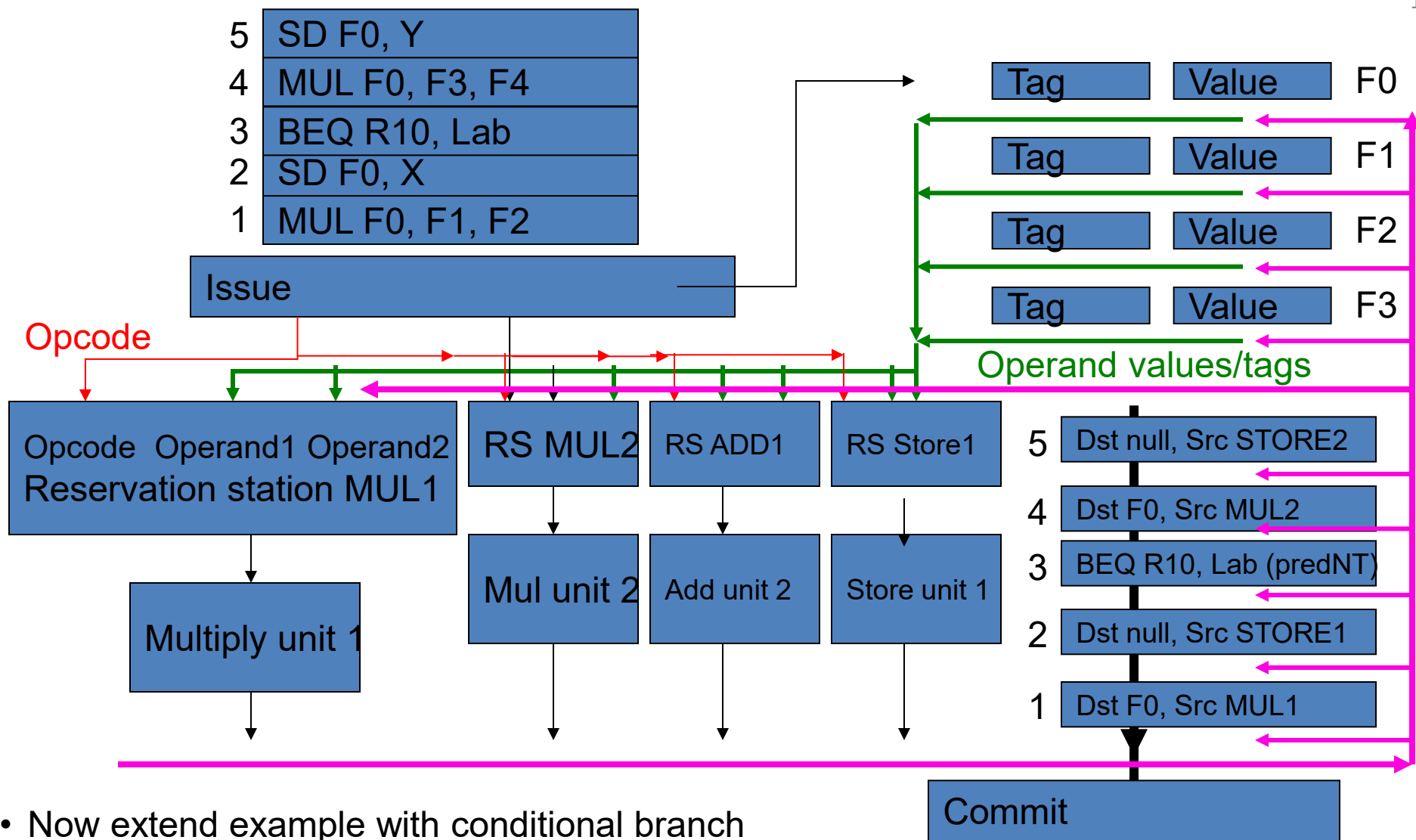
Common data bus

Commit

F0	value
F1	value
F2	value
F3	value

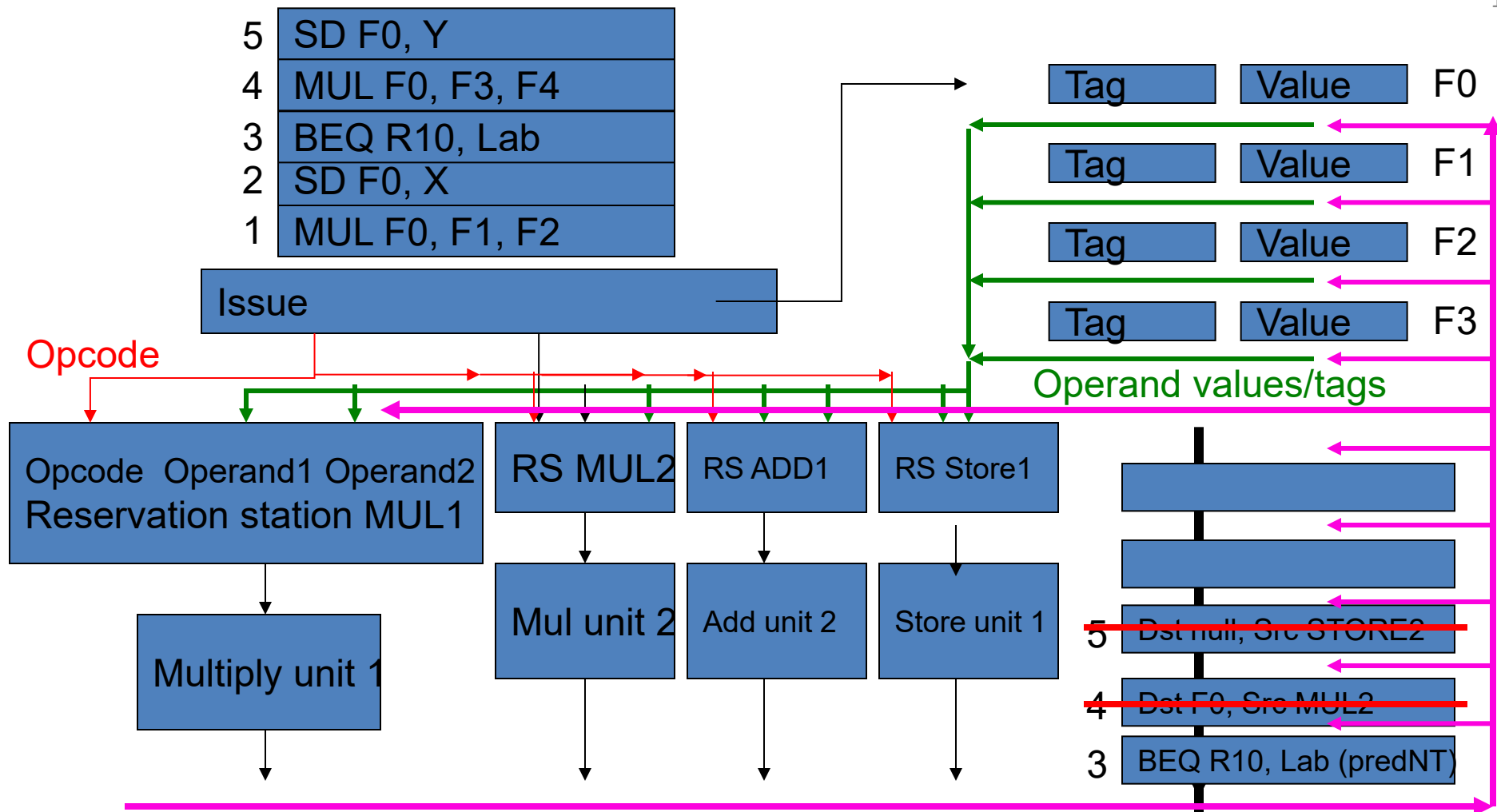
Issue-side registers
(updated speculatively)

Commit-side registers
(updated when speculation resolved)

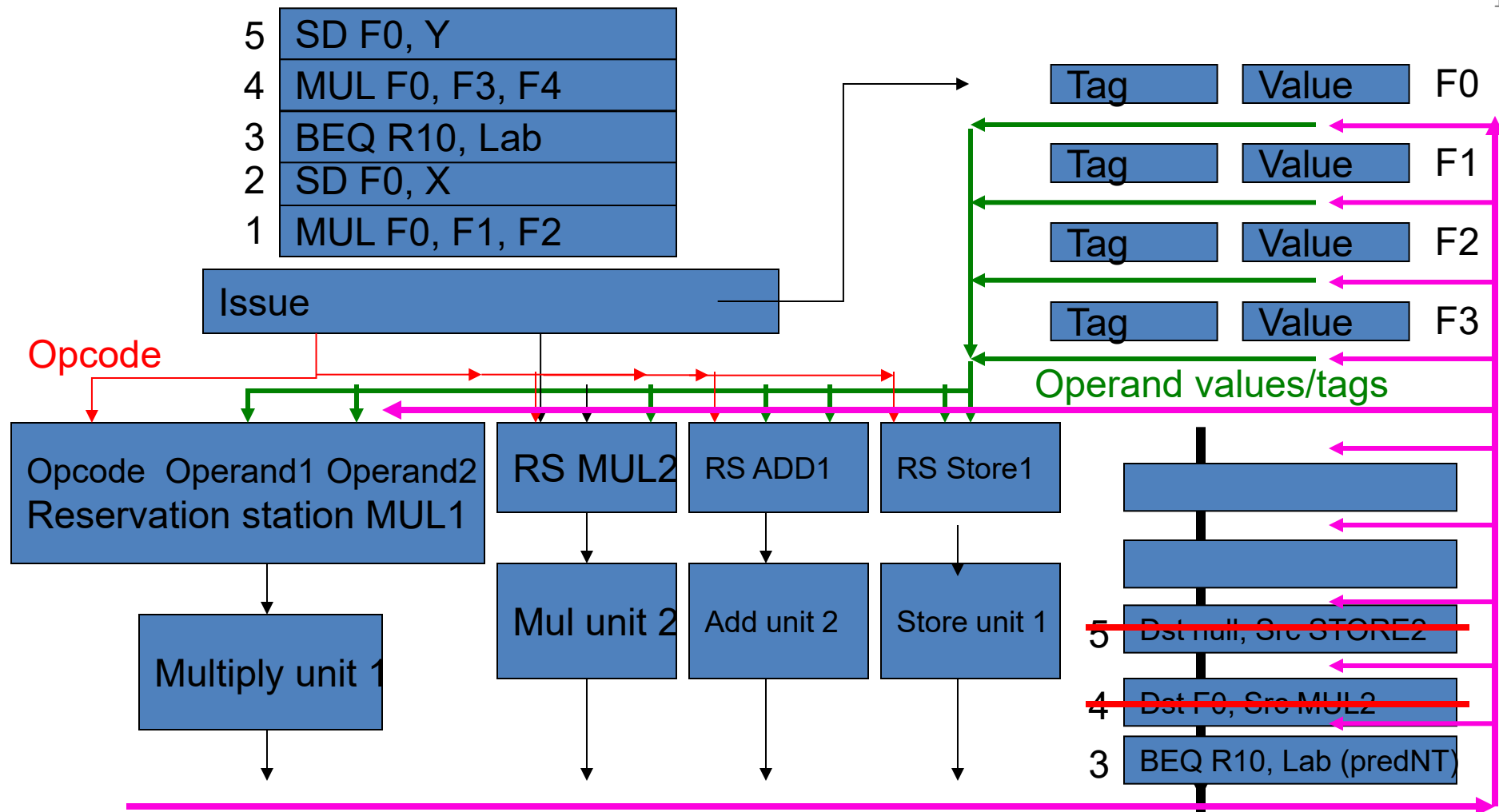


- Now extend example with conditional branch
- Assume predicted Not Taken
- When BEQ reaches head of commit queue, all instructions which have been issued but have not yet committed are erroneous

F0	value
F1	value
F2	value
F3	value



- 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

F0	Value from MUL1
F1	value
F2	value
F3	value

Some subtleties to think about...

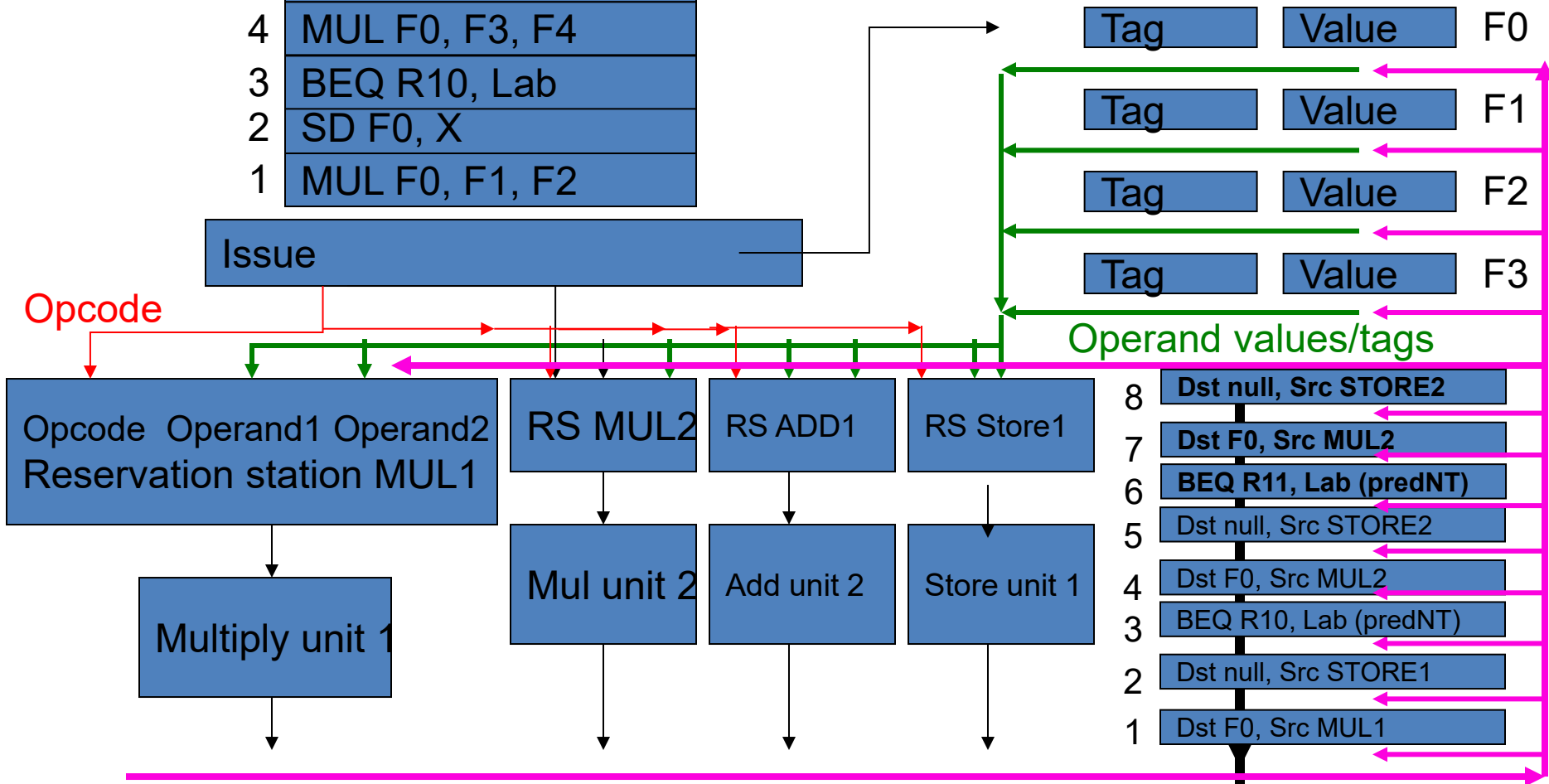
- It's vital to reduce the branch misprediction penalty. Does the Tomasulo+ROB scheme described here roll-back as soon as the branch is found to be mispredicted?
- This discussion has assumed a single-issue machine. How can these ideas be extended to allow multiple instructions to be issued per cycle?
 - 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

8	SD F0, Y
7	MUL F0, F3, F4
6	BEQ R11, Lab
5	SD F0, Y
4	MUL F0, F3, F4
3	BEQ R10, Lab
2	SD F0, X
1	MUL F0, F1, F2



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

F0	value
F1	value
F2	value
F3	value

Speculating across more than one branch

8	SD F0, Y
7	MUL F0, F3, F4
6	BEQ R11, Lab
5	SD F0, Y
4	MUL F0, F3, F4
3	BEQ R10, Lab
2	SD F0, X
1	MUL F0, F1, F2

Issue

Opcode

Opcode	Operand1	Operand2
Reservation station	MUL1	

Multiply unit 1

RS MUL2

Mul unit 2

RS ADD1

Add unit 2

RS Store1

Store unit 1

Tag	Value	
		F0
		F1
		F2
		F3

Operand values/tags

8	Dst null, Src STORE2
7	Dst F0, Src MUL2
6	BEQ R11, Lab (predNT)
5	Dst null, Src STORE2
4	Dst F0, Src MUL2
3	BEQ R10, Lab (predNT)

Commit

F0	value
F1	value
F2	value
F3	value

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

Speculating across more than one branch

8	SD F0, Y
7	MUL F0, F3, F4
6	BEQ R11, Lab
5	SD F0, Y
4	MUL F0, F3, F4
3	BEQ R10, Lab
2	SD F0, X
1	MUL F0, F1, F2

Issue

Opcode

Tag	Value	F0
Tag	Value	F1
Tag	Value	F2
Tag	Value	F3

Operand values/tags

Opcode Operand1 Operand2
Reservation station MUL1

RS MUL2

RS ADD1

RS Store1

Multiply unit 1

Mul unit 2

Add unit 2

Store unit 1

8	Dst null, Src STORE2
7	Dst F0, Src MUL2
6	BEQ R11, Lab (predNT)
5	Dst null, Src STORE2
4	Dst F0, Src MUL2
3	BEQ R10, Lab (predNT)

Commit

F0
F1
F2
F3

value
value
value
value

- 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:

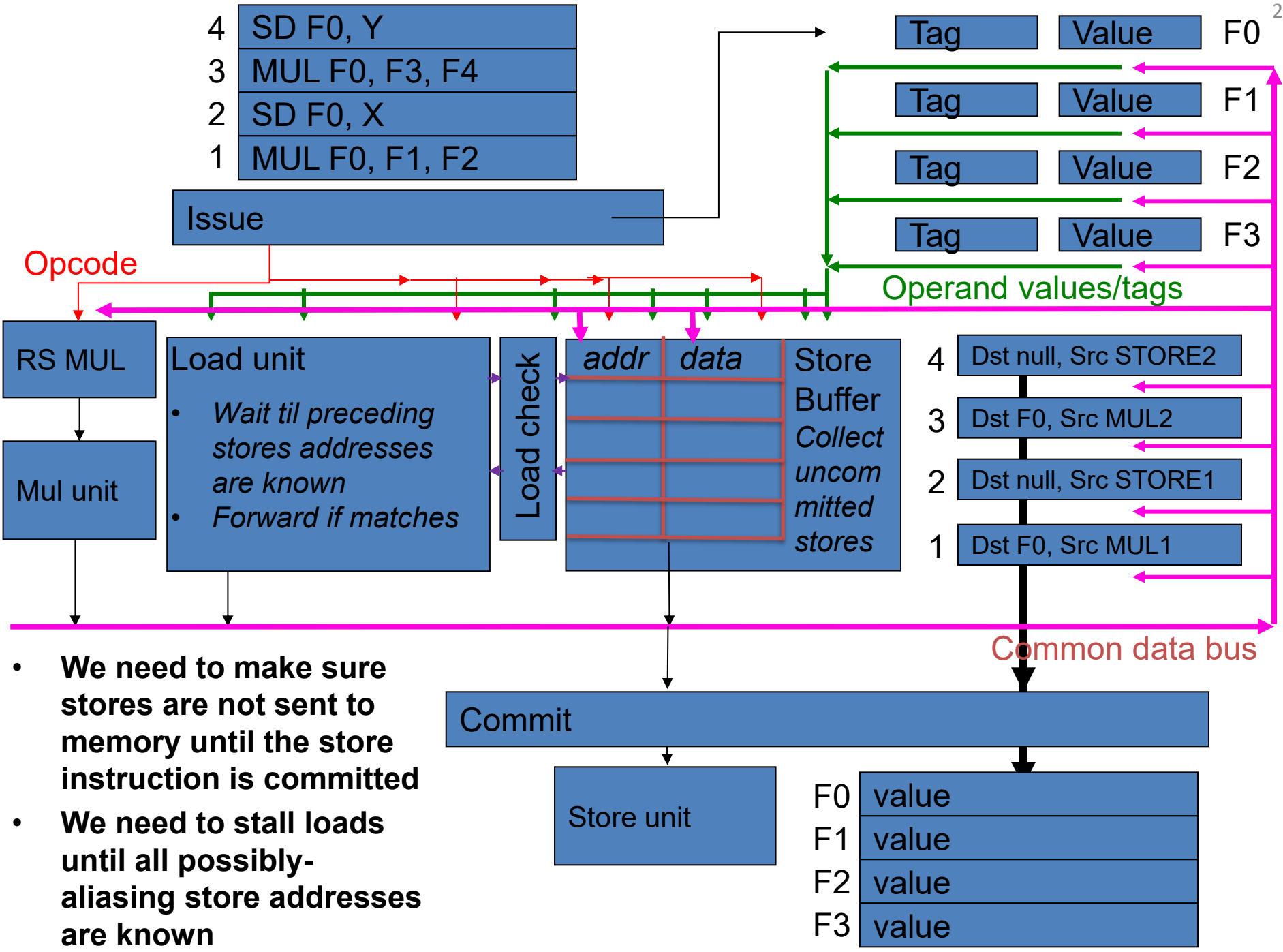
Shen and Lipasti “Modern Processor Design” pg 271, or

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 F2 from address R3

- Can we (should we?) forward F0 from *i1* to *i4*?
- What if R2=R2?
- 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)*

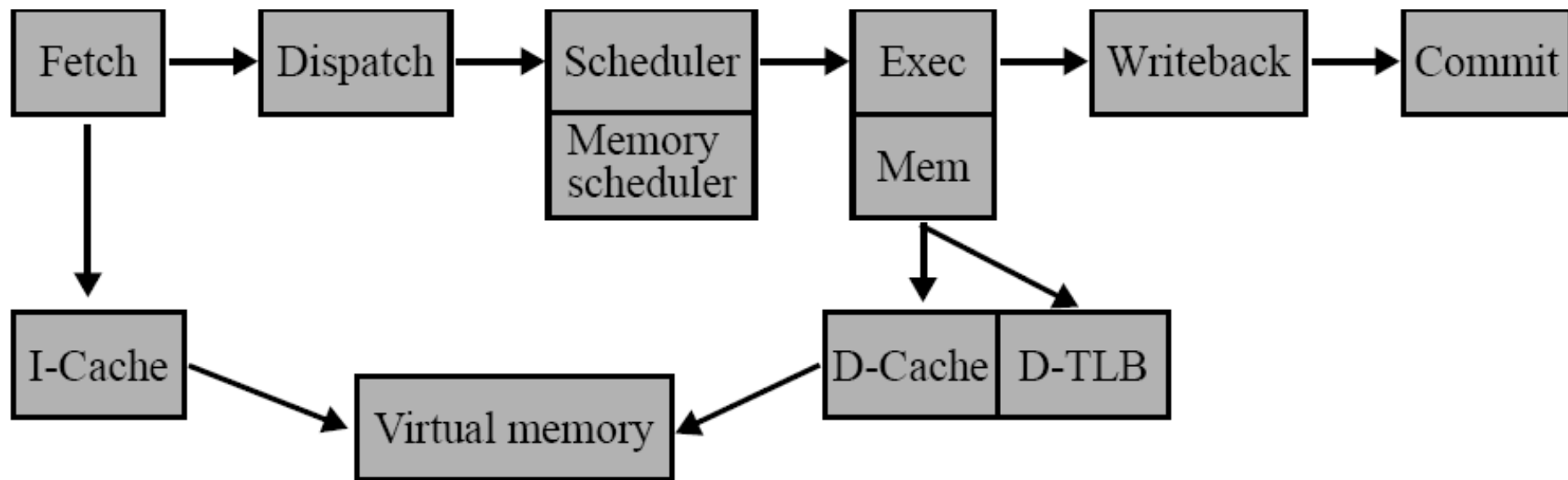
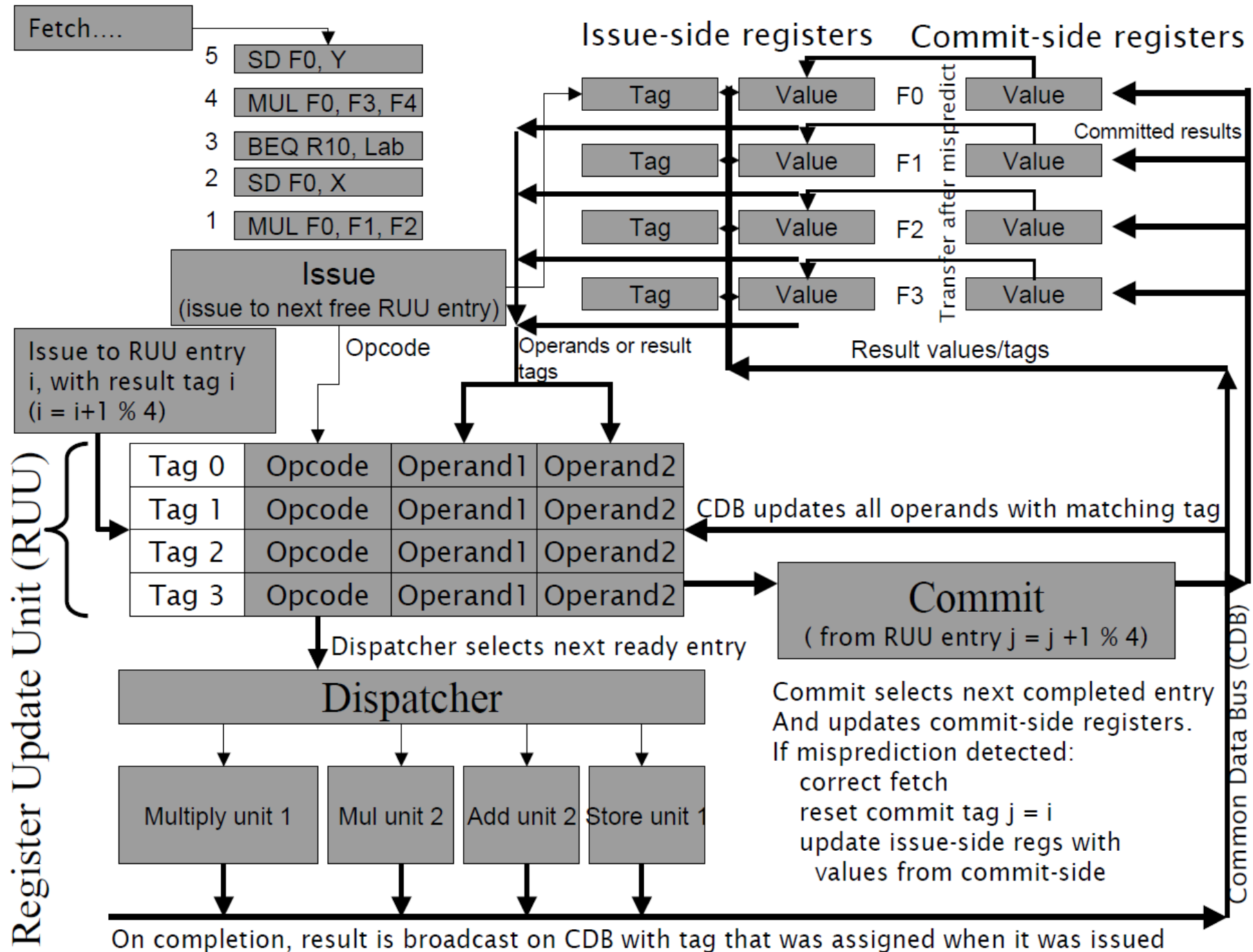
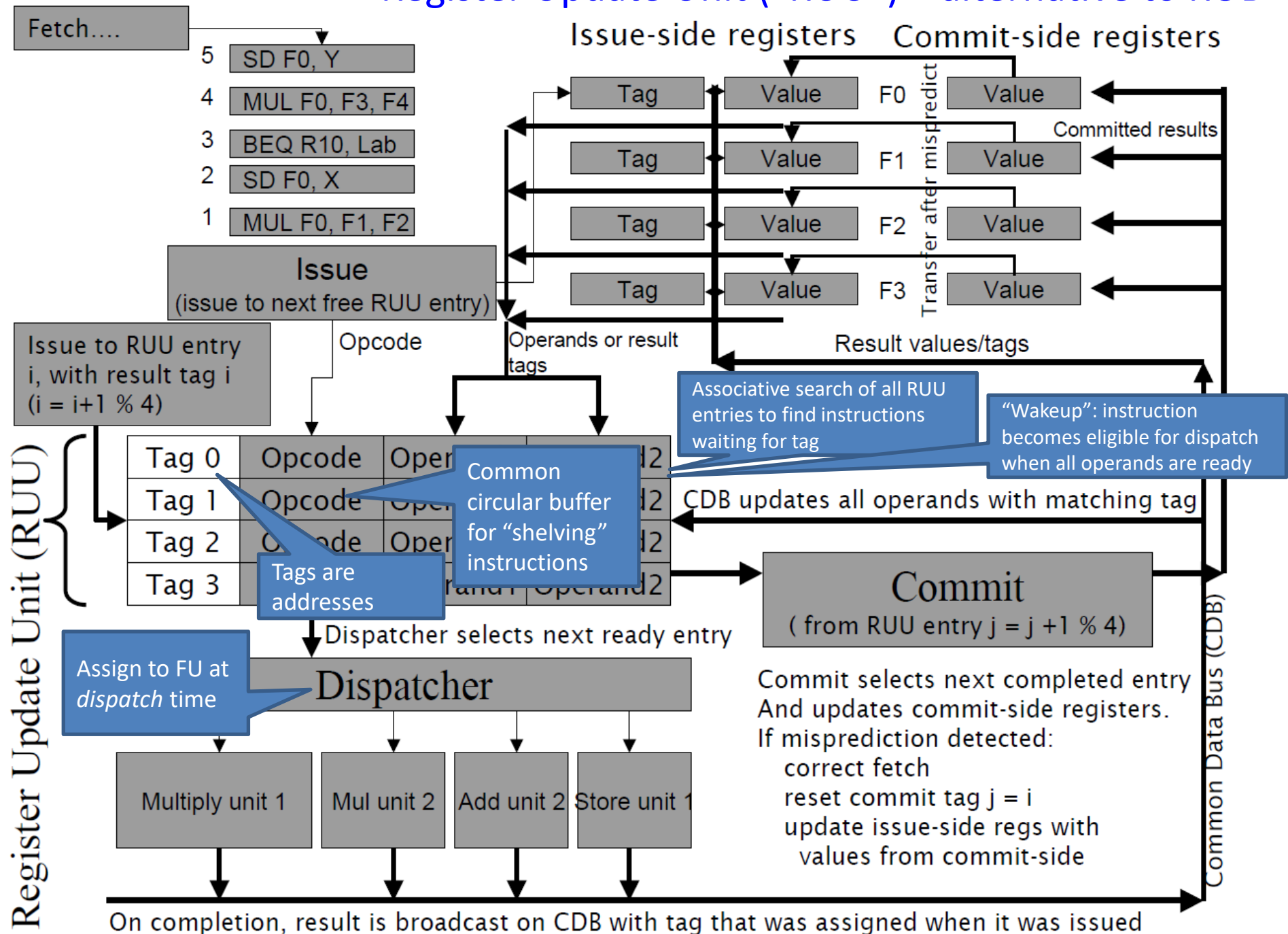


Figure 5. Pipeline for sim-outorder

- 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

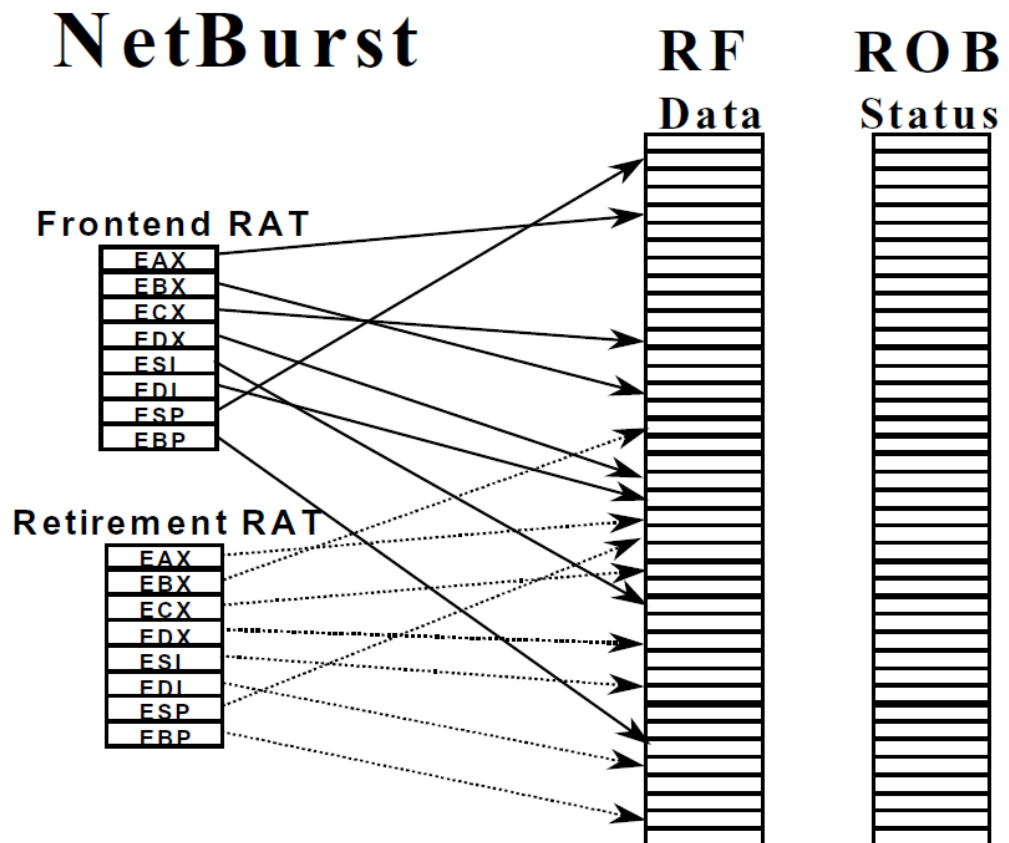
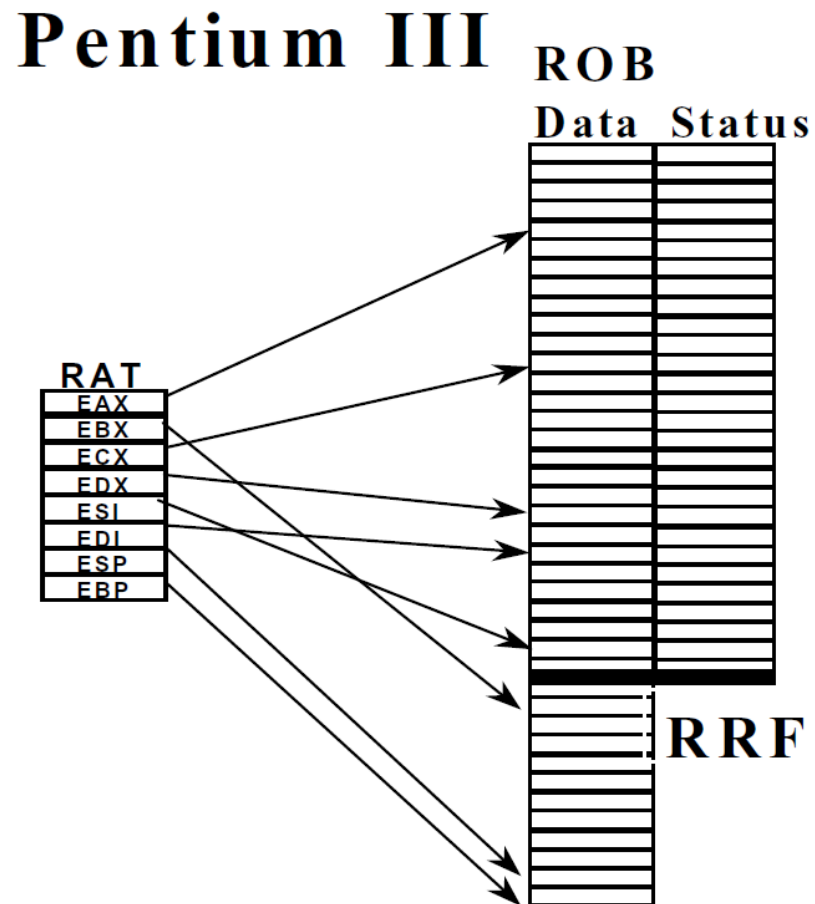


Register Update Unit ("RUU") – alternative to ROB 32



RUU vs ROB

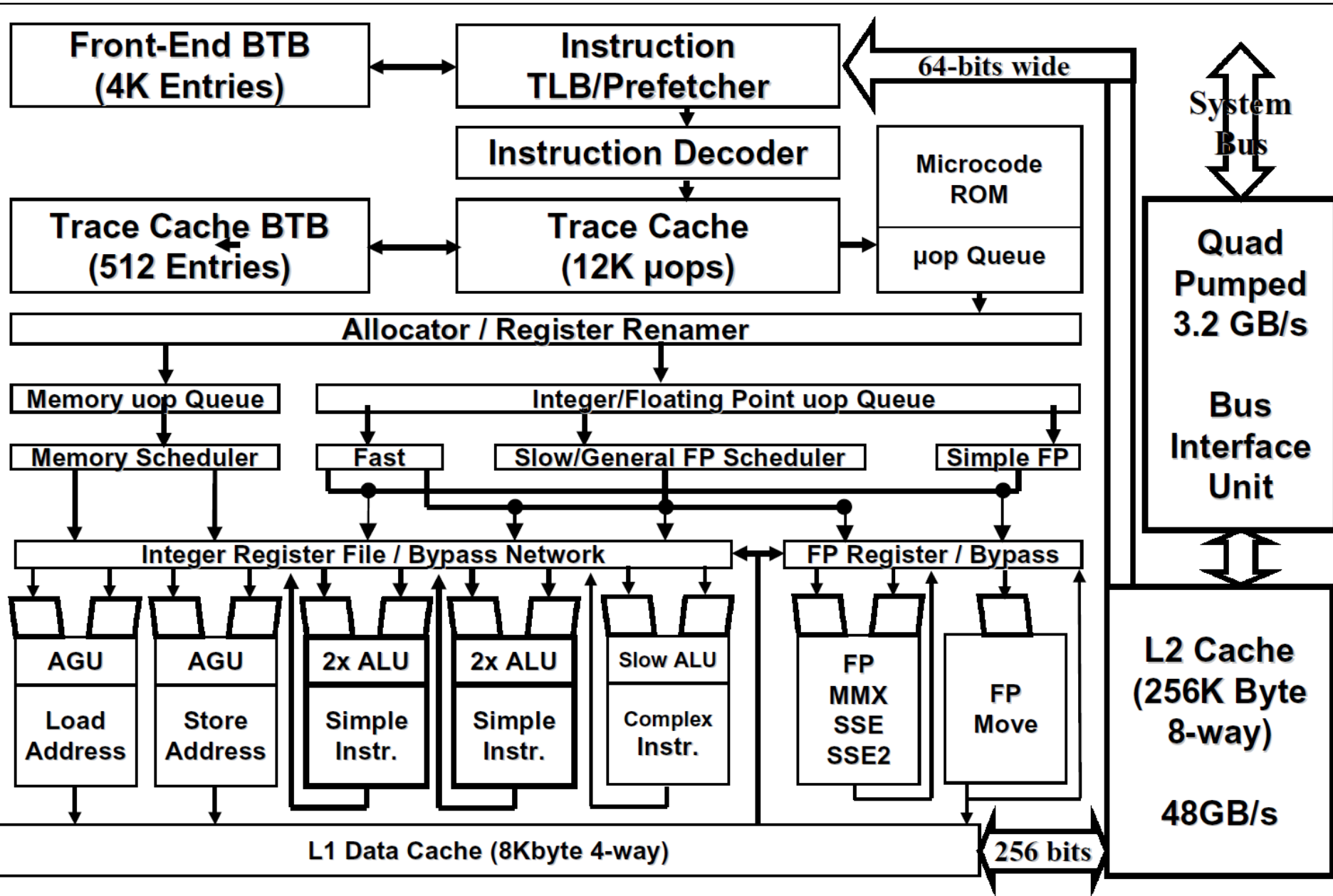
- In the Tomasulo+ROB design shown in these slides, registers *and* ROB entries have a tag
 - Every register, ROB entry and reservation station needs a comparator to monitor the CDB
- With the RUU, the tags *are* the ROB entry numbers
 - The ROB entry serves as a renamed register for its instruction's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

- 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

Q: How are registers allocated and freed?



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;
}
```

**Real
example**

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
```

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
- Online book:
 - See Chapters 2-5 of Sarangi's book/course at <https://www.cse.iitd.ac.in/~srsarangi/advbook/index.html>
- 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:
 - Gem5: <http://www.gem5.org>
 - SimEng: <https://uob-hpc.github.io/SimEng/>
 - Zsim: <https://github.com/s5z/zsim>, SimpleScalar: www.simplescalar.com/, Liberty: <http://liberty.cs.princeton.edu/>, SimFlex: <http://parsa.epfl.ch/simflex/>, SIMICS: <http://www.windriver.com/products/simics/>
 - Deep learning-based approaches including Tao: <https://dl.acm.org/doi/pdf/10.1145/3656012> and Scalable Deep Learning-Based Microarchitecture Simulation on GPUs <https://www.osti.gov/biblio/1989626>

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 uncommitted 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 $e2 \neq e3$. But e1, e2, e3 might take a while to be 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:

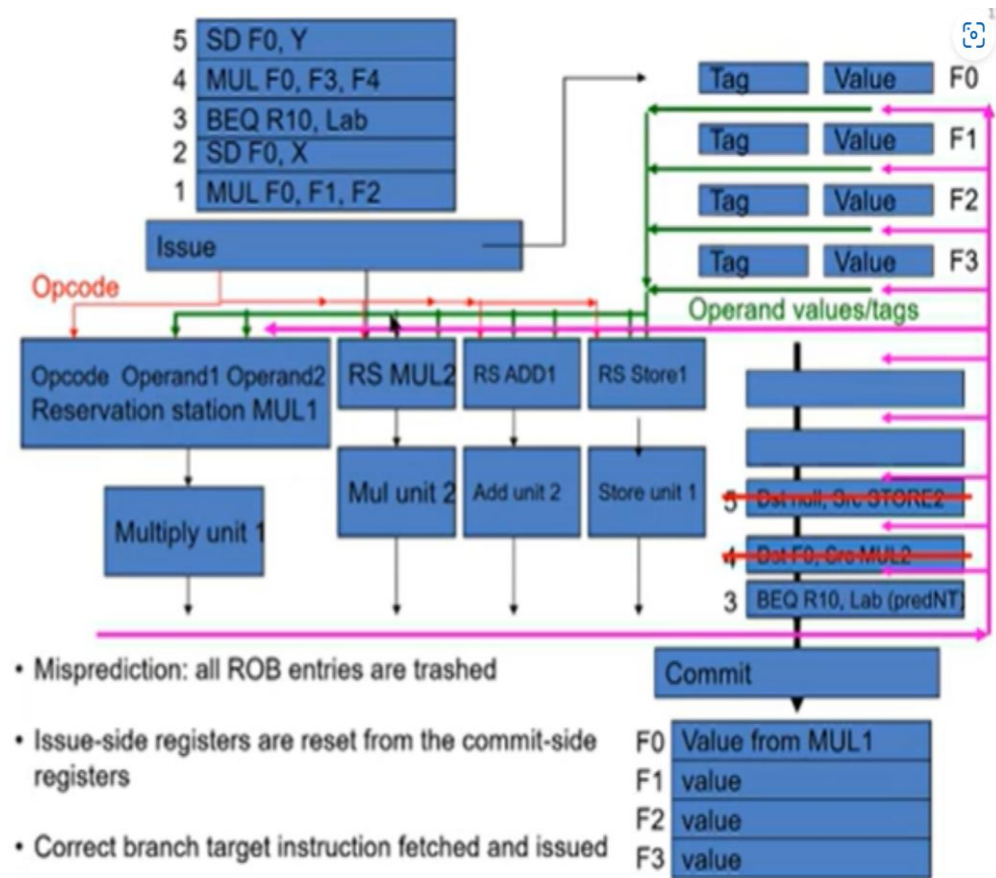
<https://elixir.bootlin.com/linux/v2.6.36/source/kernel/sched.c#L2834>

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).

Student question: context switches



When the result of instruction 3 (BEQ) is computed and verified against the speculated value in ROB:

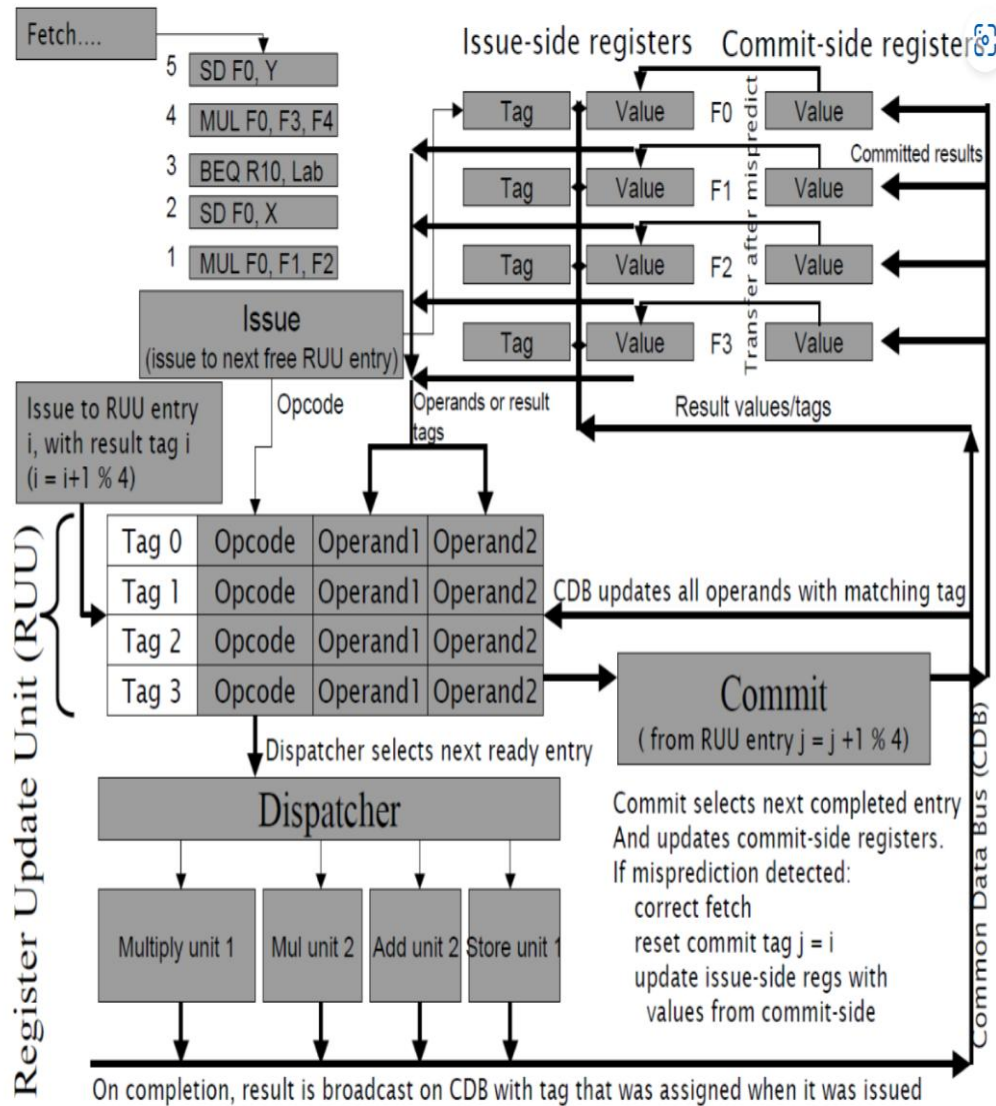
- is there a comparer hardware at this point?
- Or is there a bit that tracks whether the value of ROB for instruction 3 gets updated? If yes, then proceed with the precision interrupt mechanism?

say ROB is wrong,

- All the ROB entries above the instruction 3 (BEQ) are cleared. Does the ROB entry for instruction 3 get cleared as well? Or is the ROB entry just updated with the correct value?

- re: "Say the ROB is wrong, All the ROB entries above the instruction 3 (BEQ) are cleared. Does the ROB entry for instruction 3 get cleared as well? Or is the ROB entry just updated with the correct value?"
- I think the ROB entry for the BEQ can be cleared as part of handling the rollback for a branch misprediction. The important thing is that the correct branch target is fetched - so what we need in the ROB is in fact the instruction from the branch target, as soon as it is issued. The commit-side registers in the right state for that program point.
- re: "When the result of instruction 3 (BEQ) is computed and verified against the speculated value in ROB: Q1: is there a comparer hardware at this point? Q2: Or is there a bit that tracks whether the value of ROB for instruction 3 gets updated? If yes, then proceed with the precision interrupt mechanism?"
- The idea in the slide-ware design being presented here is that:
- The ROB entry holding the BEQ instruction monitors the CDB, and collects the branch condition as soon as it is calculated
- The Commit unit processes each ROB entry as it reaches the head of the queue. So the BEQ instruction might not get there right away - there might be other instructions that have to be committed first. In fact in the worst case, the BEQ could be delayed for a *long* time. [[example: the instruction just before the BEQ might be a LOAD that misses in the cache, but its result must be committed to the commit-side registers before we can handle the BEQ]].
- I think the point of the question is that we might know that the misprediction has happened *before* the BEQ gets to the head of the ROB.
- And you are right. But it's pretty hard to avoid the problem.
- So there is a substantial research literature on how to do it - the dominant concern is how to do it with manageable complexity.
- So this paper proposes replacing the ROB with a "Validation Buffer": [seeetoc-070726.dvi \(upv.es\)](http://www.seeetoc-070726.dvi.upv.es).
- This nice paper quantifies the performance advantage that might be achieved if you could do it: comp.nus.edu.sg/~tcarlson/pdfs/alipour2017etplooc.pdf

Student question: RUU contents



Each entry of the ROB here consists of:

The opcode - so that when the instruction is dispatched, we know what to do

Operand1 and Operand2: as shown with Tomasulo, what happens is that at issue-time, we look at the issue-side registers for the operands of the MUL instruction, F1 and F2. If the register is marked as valid, we copy the value from there into the RUU. If the value is not ready, we copy the tag - because the value is being calculated by some instruction in that has been issued but not yet committed.

We update the issue-side F0 with a tag. A good choice of tag is the index of the entry in the RUU, since that tag will not be reused until that RUU entry is committed.

Eventually the MUL is dispatched to a multiplier. Eventually this completes, and the result is broadcast on the CDB carrying that tag.

Now that the MUL has completed, its RUU entry is marked as ready-to-commit. The commit unit will commit it when that RUU entry is at the head of the commit queue. At that point, the commit unit will copy the result of the instruction into the commit-side F0.

From this we see that the RUU entry also needs to hold the result of the instruction - not shown in the diagram.

Note that we might commit the result of the MUL to the commit-side F0 long after the issue-side F0 has been overwritten by some later instruction.

What can we see in the Opcode, Operand1 and Operand2 when the instruction `MUL F0, F1, F2` is issued to the reorder buffer?

Student question: pipeline depth in OoO

“somewhat confused as to how introducing dynamic scheduling inherently increases the depth of the pipeline ? as mentioned in the dynamic scheduling summary”

Pipeline depth is, in general terms, a design choice.

With a small number of pipeline stages, you need to do a lot of work within each pipeline stage. So the achievable clock rate will probably be slower (since it depends on the maximum depth of the combinational logic between pipeline stages - as we discussed in the first pipelining lecture).

A processor with a small number of pipeline stages operates with less parallelism: we can't overlap very many successive instructions, even when hazards allow it.

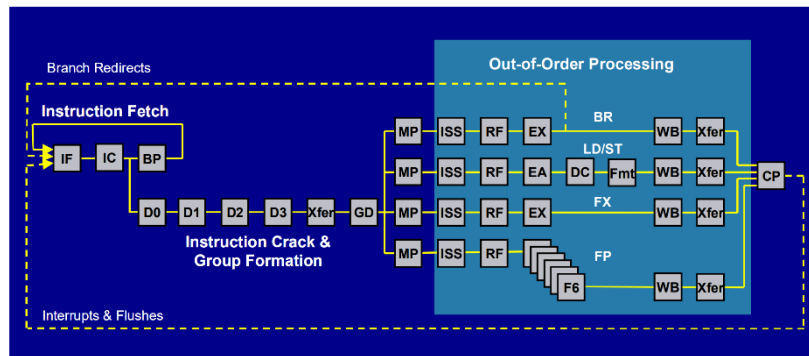
In an out-of-order processor, we can make these same tradeoffs. As you suggest, it is probably possible to build an OoO CPU with a small number of pipeline stages.

The picture looks a bit more complicated. So you could ask what is the number of pipeline stages between fetch and commit in the *best* case (perhaps a sequence of non-dependent integer adds). Some functional units might be fully pipelined but with several stages - an example might be a floating-point add or multiply. Such instructions can't be committed til later - what matters is that we can commit them at a sufficient rate. [[you might ask what happens in a simple 5-stage in-order pipeline when you add floating-point functional units - but that's a story for another day]].

So you might wonder when the pipeline depth matters.

One answer is to think about **what happens when we don't have enough instruction-level parallelism**. For example consider a loop that follows a linked list - so we have a dependent chain of loads. The critical thing here is actually not the pipeline depth - it's the forwarding latency, the chain of events from completing an operation, broadcasting the result on the CDB, collecting it, marking the ROB entry as ready, then signalling the dispatcher so that it can be sent to a functional unit for execution. To minimise this delay, we might try very hard *not* to pipeline this pathway!

Multi-issue OoO processors very often have deep pipelines, as illustrated in this diagram showing the pipeline structure of IBM's Power4 ([POWER4 White Paper \(ibm.com\)](#)):



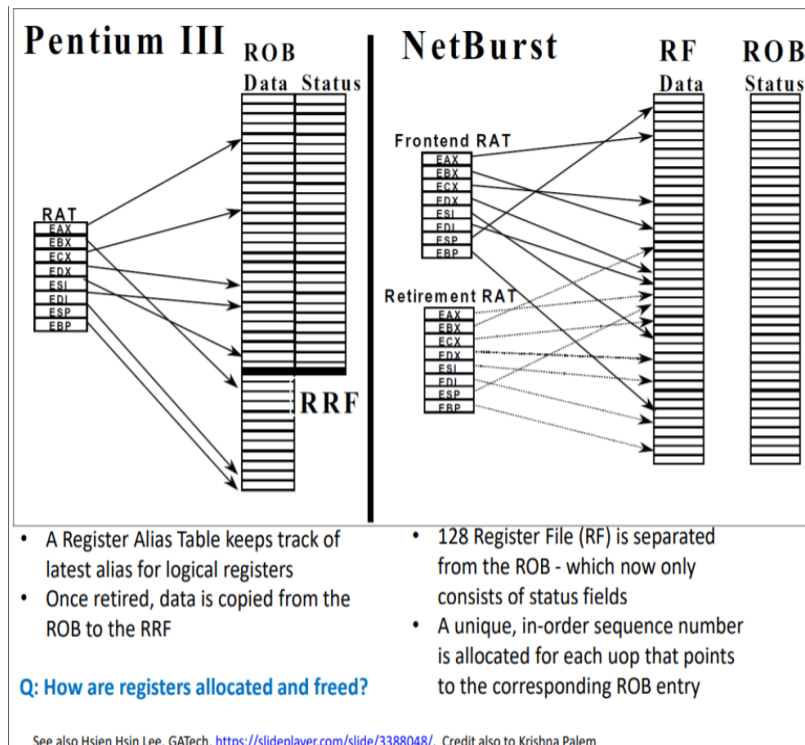
Instructions are received from the L1 instruction cache in cycle D0. "Cracking" is the process of decoding instructions into the microoperations ("uops") executed by the OoO core - some instructions may be expanded into two uops (eg a store might be split into a part to compute the address, and a second uop to collect the value to be stored). Sometimes IBM Power processors fuse two adjacent instructions into one uop. Group formation is where the dependence between uops is determined, in order to decide which can be issued together.

You might wonder why you would care about having lots of stages *before* issue - after all it doesn't affect the forwarding pathway discussed above.

But it does matter when we rollback for a misprediction. It increases the delay between fetching the right instruction, and starting to execute it. So it affects the branch misprediction penalty.

Student question: Physical Register File

For the NetBurst architecture:



- This is a nice question because it exposes the complexity of the Netburst design - which is common to many other architectures. Physical Registers (PRs) are allocated in the Physical Register File (PRF) *dynamically*, and released for re-allocation only when no references exist.
- For details of the implementation, see, for example, the documentation for SonicBOOM: [The Rename Stage — RISC-V-BOOM documentation \(boom-core.org\)](https://www.riscv-boom.co.uk/docs/boom-core.org)
- You might enjoy this footnote from there:
- "Conceptually, branches are often described as "snapshotting" the Free List (along with an *OR'ing* with the current Free List at the time of the misprediction). However, snapshotting fails to account for physical registers that were allocated when the snapshot occurs, then become freed, then becomes re-allocated before the branch mispredict is detected. In this scenario, the physical register gets leaked, as neither the snapshot nor the current Free List know that it had been freed. Eventually, the processor slows as it struggles to maintain enough inflight physical registers, until finally the machine comes to a halt. If this sounds autobiographical because the original author (Chris) may have trusted computer architecture lectures, well..."

We update the pointers for the retirement RAT based on the Frontend RAT once the ROB indicates the instruction can be committed. What happens if we rewrite the contents of a register already assigned to a Retirement RAT by a new Frontend RAT but hit a branch misprediction so we end up losing the previously committed value. This can happen if there exists some delay between updating a register's contents and committing the register by creating a pointer in the Retirement RAT.

Student question: partial rollback

Q: In the lecture on speculation in dynamically scheduled processors, you mention that when there is a branch misprediction, all ROB entries are trashed. However, in cases where there are short if-else sections that do not contain many instructions, flushing the ROB would mean flushing a lot of instructions after the conditional that are not dependent on the conditional either. Is there any way of only flushing the instructions that are in / dependent on a specific branch?

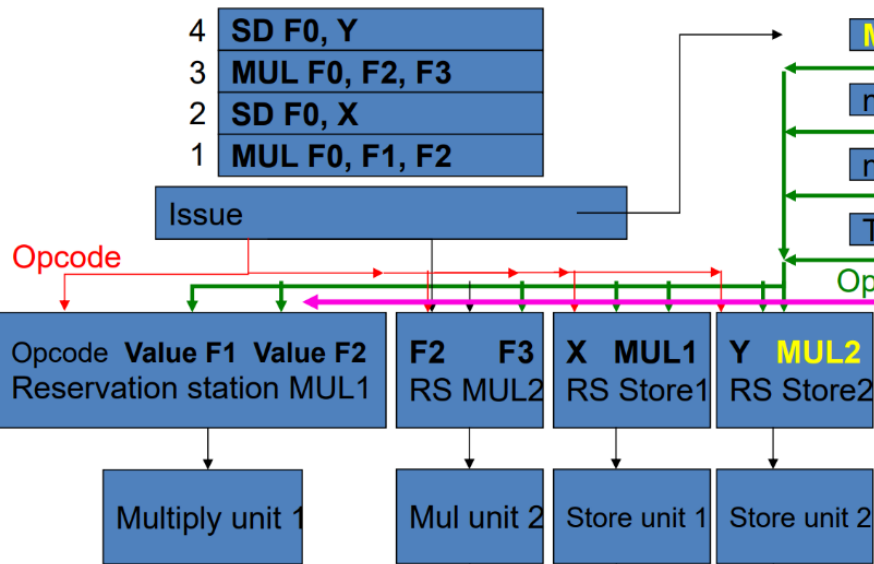
I also read online that modern processors can have ROBAs that execute instructions from separate paths in the control flow at once, mitigating the impact of mispredicted branches, if so, once the branch has been calculated, I assume the entire ROB does not get flushed, so how do we ignore certain instructions whilst including others?

A:

- This has indeed been an area of some research, going back a long way.
- Examples include:
- Dynamic Instruction Reuse Avinash Sodani and Gurindar S. Sohi
<https://ftp.cs.wisc.edu/sohi/papers/1997/isca.inst-reuse.pdf>
- Reducing Branch Misprediction Penalties Via Dynamic Control Independence Detection Yuan Chou, Jason Fung, and John P. Shen
<https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=876032bf2c0e0394946af9054014c1761d80e1e7>
- The performance gain is useful but I think not revolutionary. One reason is this, I think: what we're interested in are the instructions after a mispredicted if-then block that are redundantly re-executed.
- If they're pure arithmetic they'll be re-executed very fast. The interesting case is if they're loads that miss in L1. But when we re-execute those loads, they will likely now be hits (or at least arriving soon).

Student question: partial rollback

Q:



A:

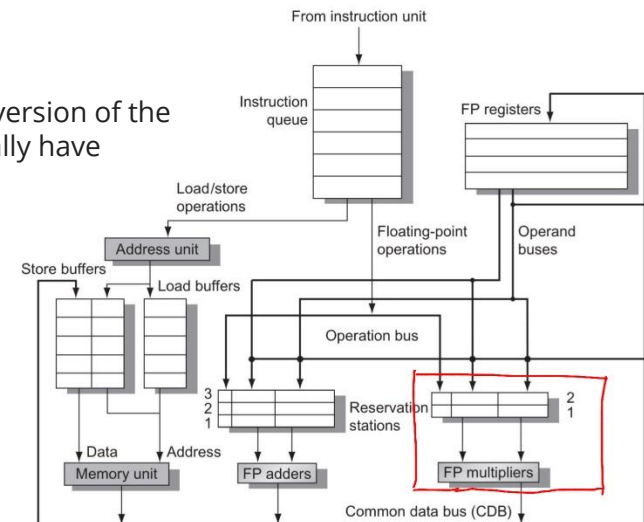
- You could indeed have several reservation stations for each functional unit. Note that when a result is ready it needs to be broadcast on the CDB with the tag of the reservation station it was issued to, not the functional unit.
- A similar consideration would apply if (as is normal) the FU is pipelined.

Lets say there was another MUL instruction between 1-2 which is put on RS MUL 2. what happens then to the other MUL instruction that is coming after (3)?

ie. can the RS hold more than one instruction or is it stalled in the issue state of this process.

Further comment:

- The Hennessy and Patterson version of the Tomasulo diagram does actually have multiple RS's per FU



Student question: Scoreboard



- Reservation stations
 - Permit instruction issue to advance past integer control flow operations
 - Also buffer old values of registers - totally avoiding the WAR stall (in contrast with a "scoreboard" design that doesn't do register renaming).

Hi, please could you shed some more light on the "scoreboard" design? I've seen the name used around as well.



The scoreboard idea is not covered in the course because I figured you don't need to know about it.



Hennessy and Patterson summarise it nicely:

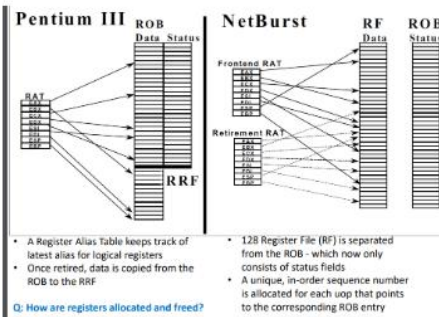
"It is important to remember that the tags in the Tomasulo scheme refer to the buffer or unit that will produce a result; the register names are discarded when an instruction issues to a reservation station. (This is a key difference between Tomasulo's scheme and scoreboarding: In scoreboarding, operands stay in the registers and are only read after the producing instruction completes and the consuming instruction is ready to execute." (6th edition pg175).

They provide a detailed explanation in Appendix C, pp C-71.

You can find out more here: [Scoreboarding - Wikipedia](#)

See also Sarangi's book section 5.6.5 ([nextgen-comp-arch.pdf](#) pg 203), which discusses some implementation details in the context of Itanium. While Itanium is an in-order pipeline, the scoreboard keeps track of which registers each instruction needs, and therefore where forwarding and stalls are required.

The idea of a scoreboard also comes up in H&P 6th ed pp294, where it refers to the data structure maintained by a GPU's warp scheduler, to determine which warps are ready-to-run. The principle is the same - it tracks for which instructions the operands are already in registers.



I don't understand what's going on here, especially in NetBurst. I understand that having a large pool of physical registers like in Snapdragon x is useful, to reduce dependencies.

Why do we need to keep a track of this mapping? What is in the status field? Is it like if the instruction, that writes to that physical register, has completed?

Student question: Pentium 4



The figure comes from the wonderful 2001 article from Intel "The Microarchitecture of the Pentium 4 Processor" which you can find here: [Pentium4Arch.pdf](#) (or from the course, p16 of [Preface](#)). Which explains:

"The register renaming logic renames the logical IA-32 registers such as EAX onto the processors 128-entry physical register file. This allows the small, 8-entry, architecturally defined IA-32 register file to be dynamically expanded to use the 128 physical registers in the Pentium 4 processor. This renaming process removes false conflicts caused by multiple instructions creating their simultaneous but unique versions of a register such as EAX. There could be dozens of unique instances of EAX in the machine pipeline at one time. The renaming logic remembers the most current version of each register, such as EAX, in the Register Alias Table (RAT) so that a new instruction coming down the pipeline can know where to get the correct current instance of each of its input operand registers."

The status field does, as you say, track whether the instruction has completed. Actually it tracks a bit more. For example in the open-source SonicBOOM architecture, the status held by the ROB is documented here: [The Reorder Buffer \(ROB\) and the Dispatch Stage — RISCv-BOOM documentation](#) :

- is entry valid?
- is entry busy?
- is entry an exception?
- branch mask (which branches is this entry still speculated under?)
- rename state (what is the logical destination and the stale physical destination?)
- floating-point status updates
- other miscellaneous data (e.g., helpful for statistic tracking)

They go on to explain that they also track the PC.

Doing wakeup more efficiently

- For more in wakeup see Sarangi's book Ch4:
 - <https://www.cse.iitd.ac.in/~srsarangi/advbook/chapters/issue-execute-commit.pdf>
- For more on how to do it without a huge number of comparators see also Henry Wong's PhD thesis:
 - <https://tspace.library.utoronto.ca/handle/1807/80713>
 - Esp pp101

Student question: control flow that rejoins

- consider:

x is a parameter

y is a parameter

z is a parameter

r1, r2, r3 are registers used in later code

BEQ R1, Label

A: $r1 = x * y$

B: $r2 = x - y$

Label:

C: $r3 = x / y$ *// control flow re-joined*

D: $r4 = y \% z$ *// these values are unaffected by the code we jumped over*

- Note that control flow rejoins.
- So when we speculate the branch, we might get all the way to instruction D before discovering that the branch was mis-speculated.
- So we roll back, throwing away the work done by instructions C and D. Which we should regret, because C and D were useful correct work that is needed in the later code that uses r3 and r4.
- So we might wonder whether we should imagine a "partial rollback" scheme, where we only rollback the stuff we have to undo, and we keep the stuff that is still valid.
- I might be wrong but I don't think anyone has implemented this in a product, though there are papers, eg:
 - Reducing branch misprediction **penalty** via selective branch recovery <https://ieeexplore.ieee.org/document/1410082>
 - I quite like this Master's thesis: RingBOOM: An Implementation of a Novel High Performance Banked Microarchitecture <https://people.eecs.berkeley.edu/~krste/papers/EECS-2020-177.pdf>