COMP60001/COMP70086 Advanced Computer Architecture Chapter 2: part 1

Dynamic scheduling, out-of-order execution, register renaming (and, in part 2, speculative execution)

Hennessy and Patterson 6th ed Section 3.4 & 3.5, pp191-208

October 2025

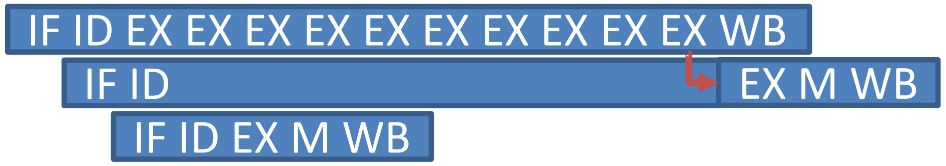
Paul H J Kelly

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)*

HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F12,F8,F14
```



- Enables out-of-order execution and allows out-of-order completion
- We will distinguish when an instruction is issued, *begins execution* and when it *completes execution*; between these two times, the instruction is *in execution*
- In a dynamically scheduled pipeline, all instructions pass through issue stage in order (in-order issue)

Data Dependence and Hazards

What constrains execution order?

#1: Instr_J is data dependent on Instr_I
Instr_J tries to read operand before Instr_I writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- or Instr_J is data dependent on Instr_K which is dependent on Instr_I
- Caused by a "True Dependence" (compiler term)
- If true dependence caused a hazard in the pipeline, called a Read After Write (RAW) hazard

Name Dependence: Anti-dependence

#2: Name dependence: when two instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name

- There are two kinds:
- Name dependence #1: anti-dependence/WAR

  Instr$_J$ writes operand *before* Instr$_I$ reads it:

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

  Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1"
- If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard

Another name Dependence: Output dependence

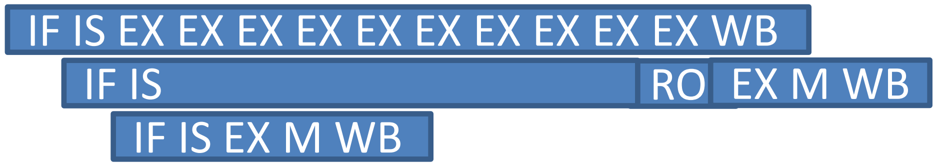
#3: Instr_J writes operand ***before*** Instr_I writes it.

$$\begin{tikzcd}[row sep=small] \text{I: sub r1,r4,r3} \arrow[d, bend right] \\ \text{J: add r1,r2,r3} \\ \text{K: mul r6,r1,r7} \end{tikzcd}$$

- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1"
- If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard

Dynamic Scheduling Step 1

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F12,F8,F14
```



- Simple pipeline had one stage to check both structural and data hazards: Instruction Decode (ID), also called Instruction Issue
- Split the ID pipe stage of simple 5-stage pipeline into 2 stages:
- *Issue:* Decode instructions, check for structural hazards
- *Read operands:* Wait until no data hazards, then read operands

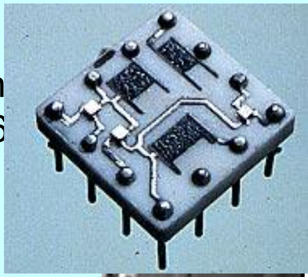
Instructions are *issued* in-order
But may stall at the Read Operands stage while others execute

Tomasulo's Algorithm

- For IBM 360/91 (before caches!)
- Goal: High Performance without special compilers
- Small number of floating point registers in the instruction set (4 in IBM 360)
  - prevented static compiler scheduling of operations
  - This led Tomasulo to try to figure out how to increase the effective number of registers — renaming in hardware!

- Why study a 1966 Computer?
- The descendents of this have flourished!
  - Alpha 21264, HP 8000, MIPS 10000/R12000, Pentium II/III/4, Core, Core2, Nehalem, Sandy Bridge, Ivy Bridge, Haswell, AMD K5,K6,Athlon, Opteron, Phenom, PowerPC 603/604/G3/G4/G5, Power 3,4,5,6, ARM A15, …

- CPU cycle time: 60 nanoseconds
- memory cycle time (to fetch and store eight bytes in parallel): 780ns
- Standard memory capacity: 2,097,152B interleaved 16 ways (magnetic cores)
- Up to 6,291,496 bytes of main storage
- Up to 16.6-million additions/second
- Ca.120K gates, ECL
- Solid Logic Technology (SLT), an IBM invention which encapsulated 5-6 transistors into a small module--a transition technology between discrete transistors and the IC
- About 12 were made

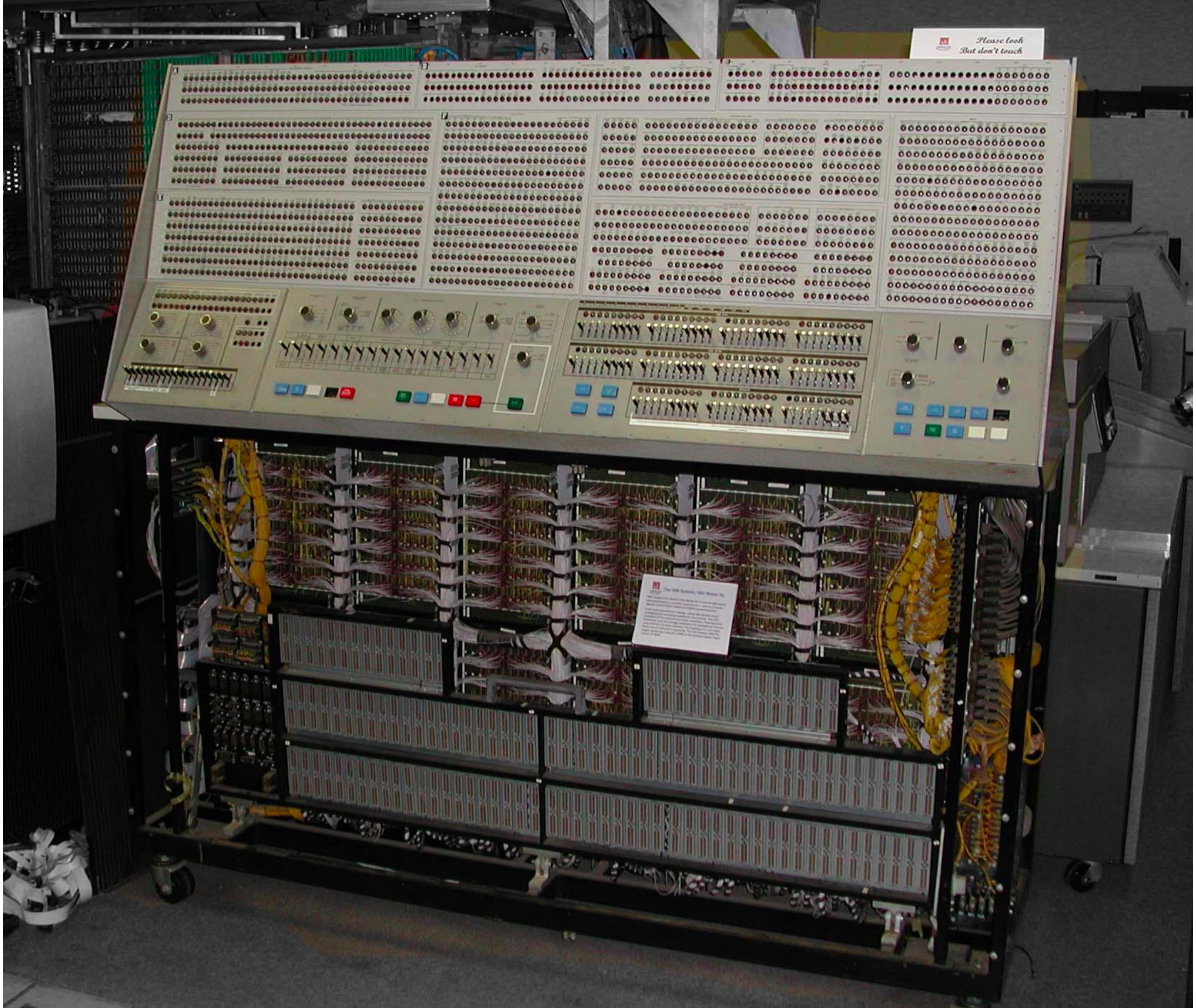


NASA Center for Computational Sciences

See:
Some Reflections on Computer Engineering: 30 Years after the IBM System 360 Model 91
Michael J. Flynn
ftp://arith.stanford.edu/tr/micro30.ps.Z

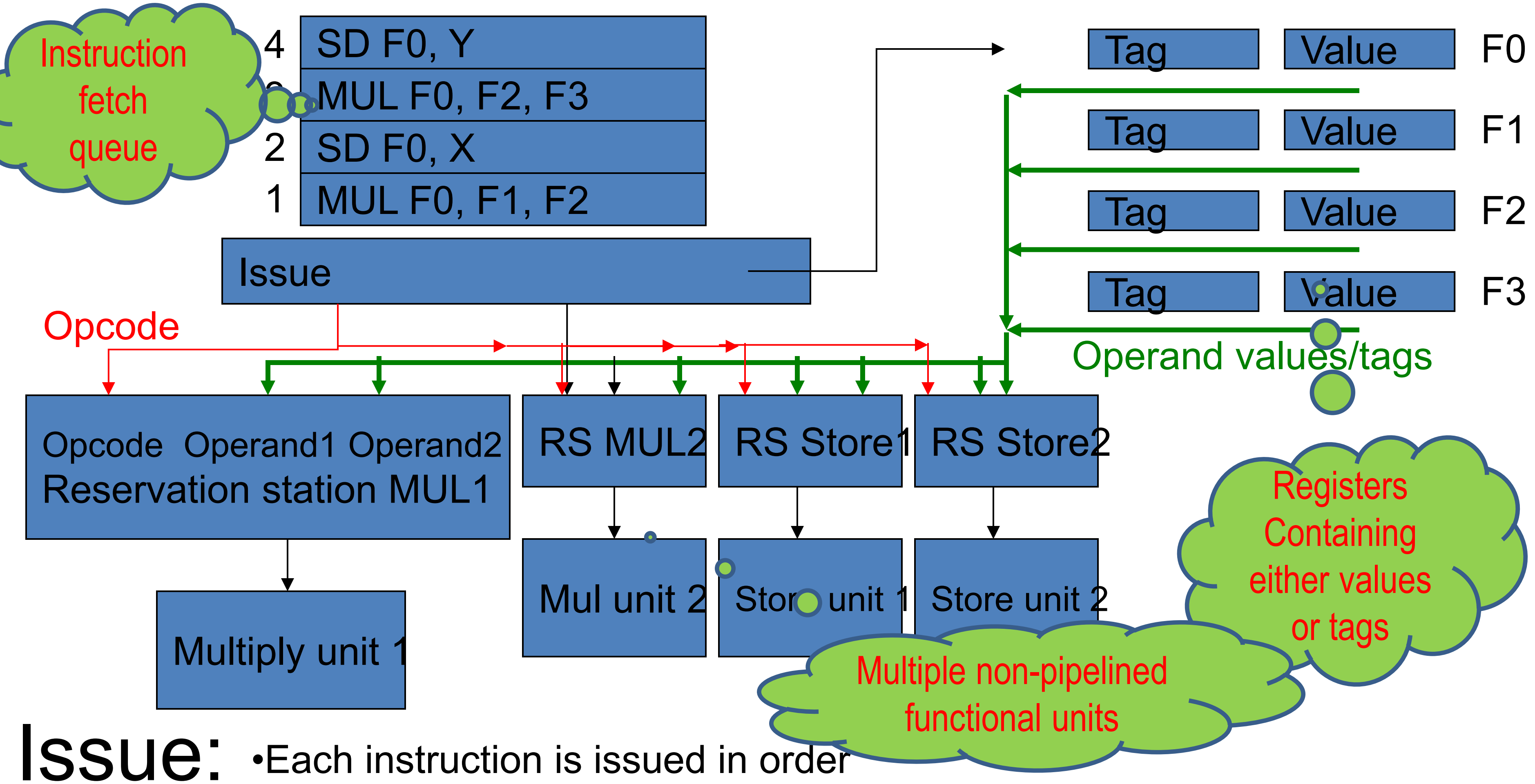
Source: http://www.columbia.edu/acis/history/36091.html

NASA's Space Flight Center in Greenbelt, Md, January 1968



http://www.columbia.edu/cu/computinghistory/36091.html

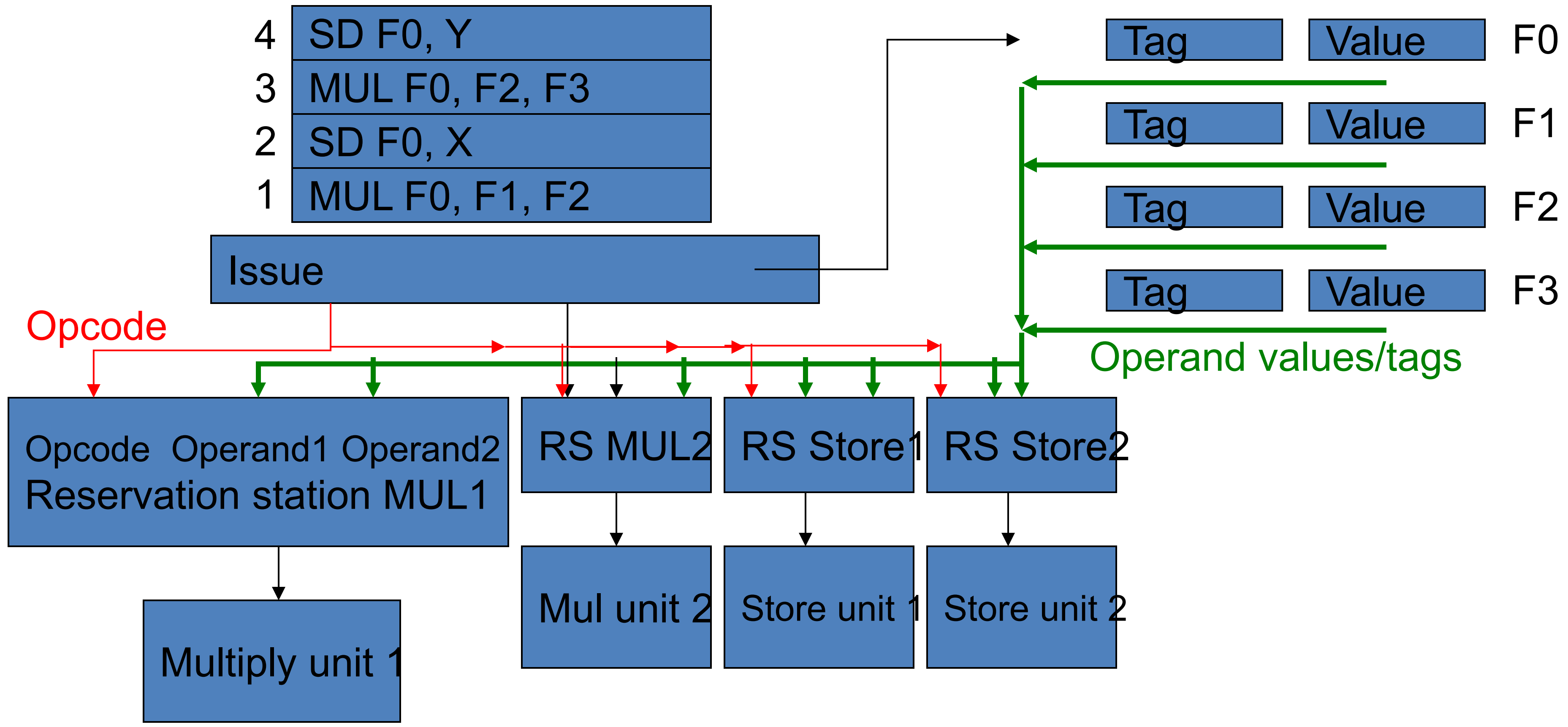
Tomasulo – closer look at instruction processing



Issue:

- Each instruction is issued in order
- Issue unit collects operands from the two instruction's source registers
- Result may be a value, or, if value will be computed by an uncompleted instruction, the tag of the RS to which it was issued.
- When instruction 1 is issued, F0 is updated to get result from MUL1
- When instruction 3 is issued, F0 is updated to get result from MUL2

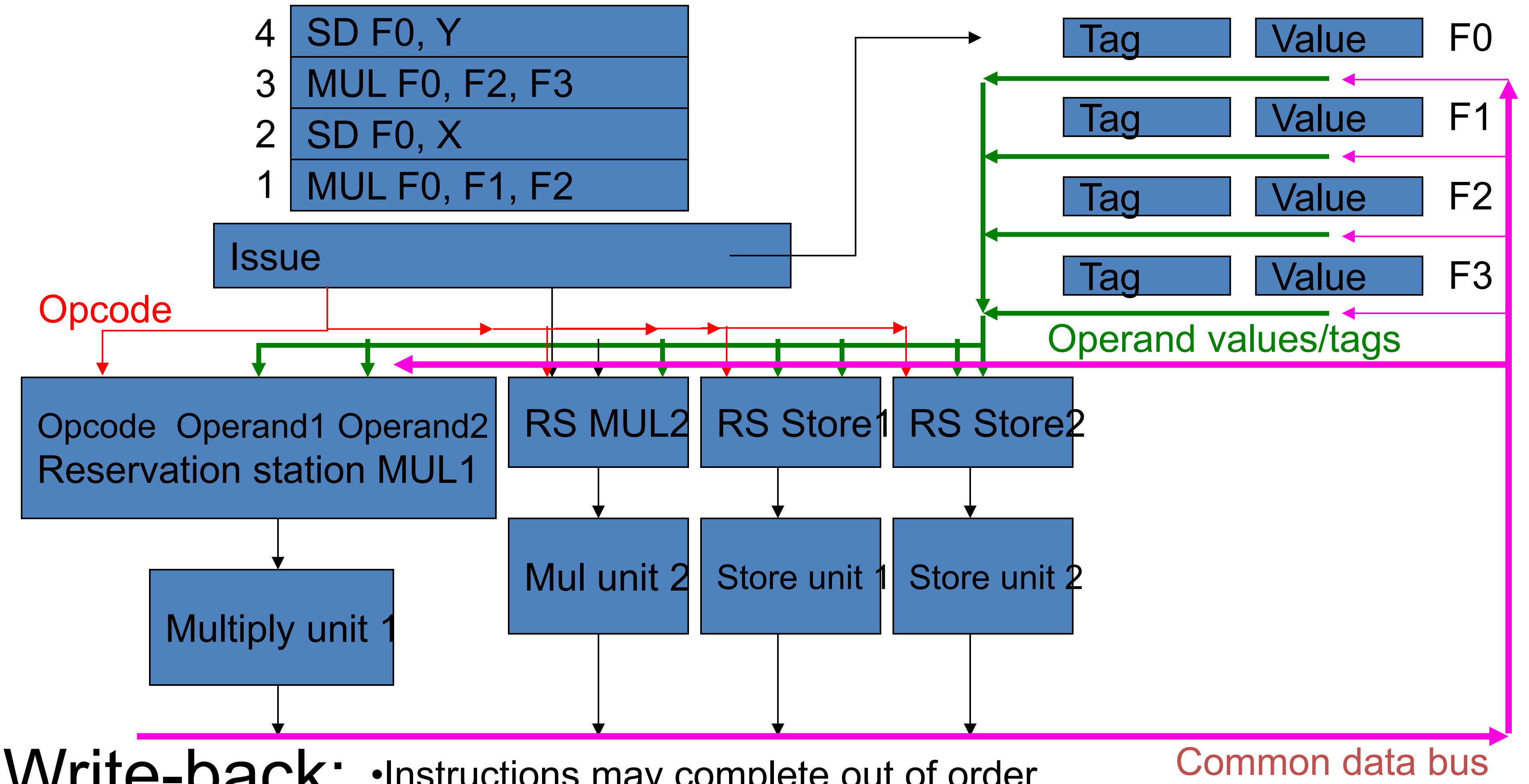
Tomasulo – closer look at instruction processing



Issue:

- Each instruction is issued in order
- Issue unit collects operands from the two instruction's source registers
- Result may be a value, or, if value will be computed by an uncompleted instruction, the tag of the RS to which it was issued.
- When instruction 1 is issued, F0 is updated to get result from MUL1
- When instruction 3 is issued, F0 is updated to get result from MUL2

Tomasulo – closer look at instruction processing

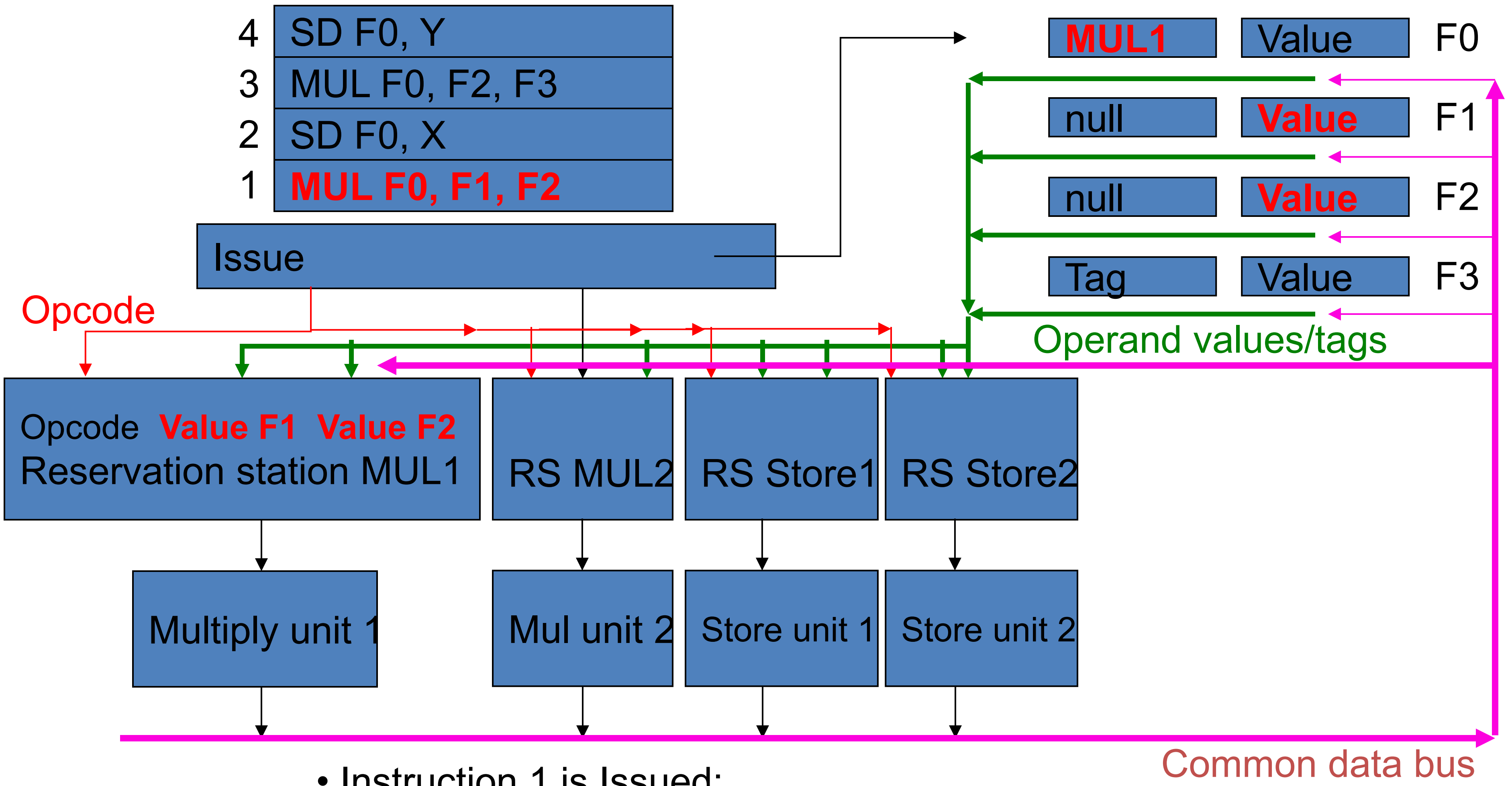


Write-back:

- •Instructions may complete out of order
- •Result is broadcast on CDB
- •Carrying tag of RS to which instruction was originally issued
- •All RSs and registers monitor CDB and collect value if tag matches
- •Any RS which has both operands and whose FU is free fires.
- •When MUL1 completes result goes to store unit but not F0

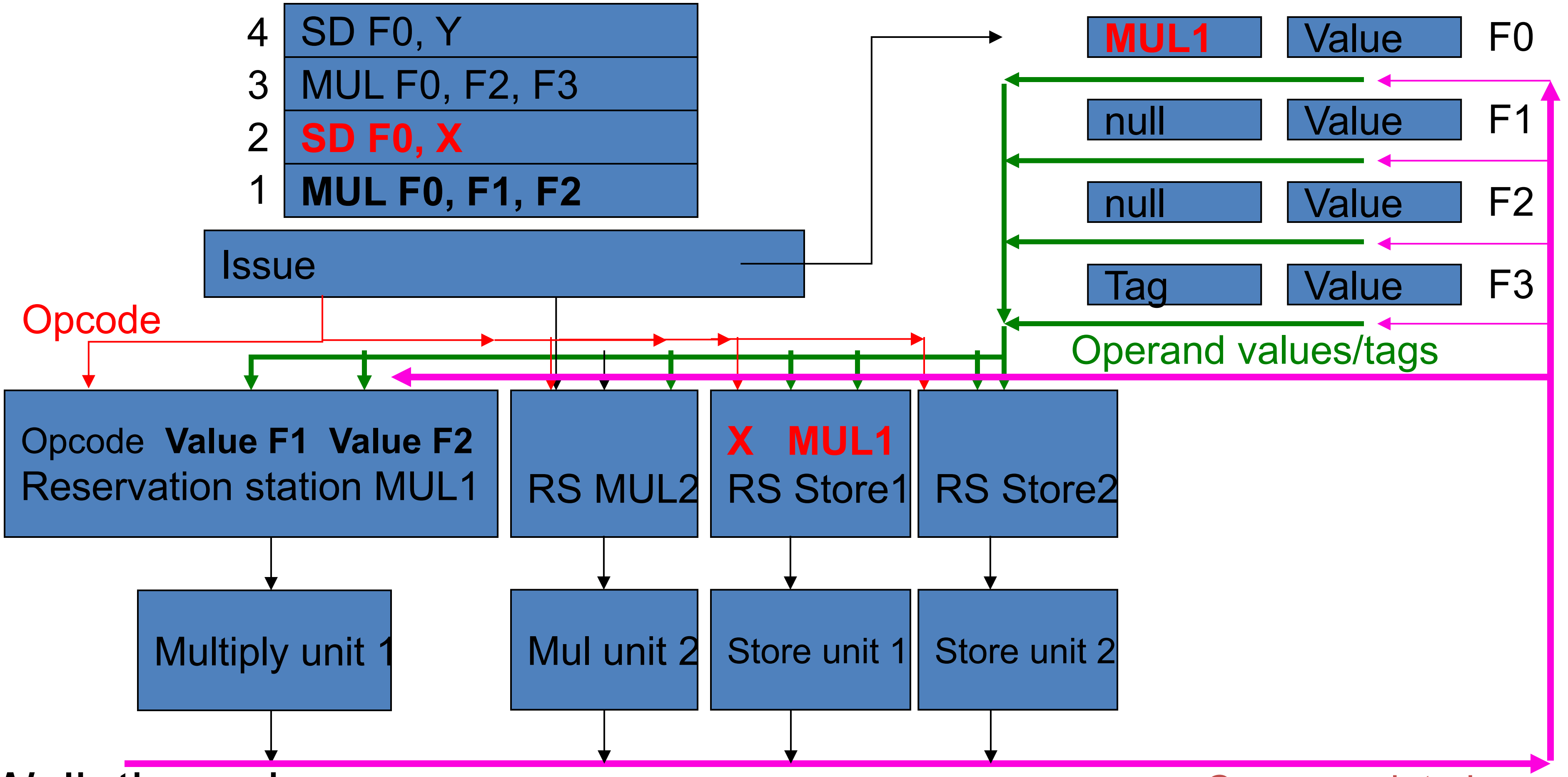
What trickery is this?

Tomasulo – **Walkthrough**



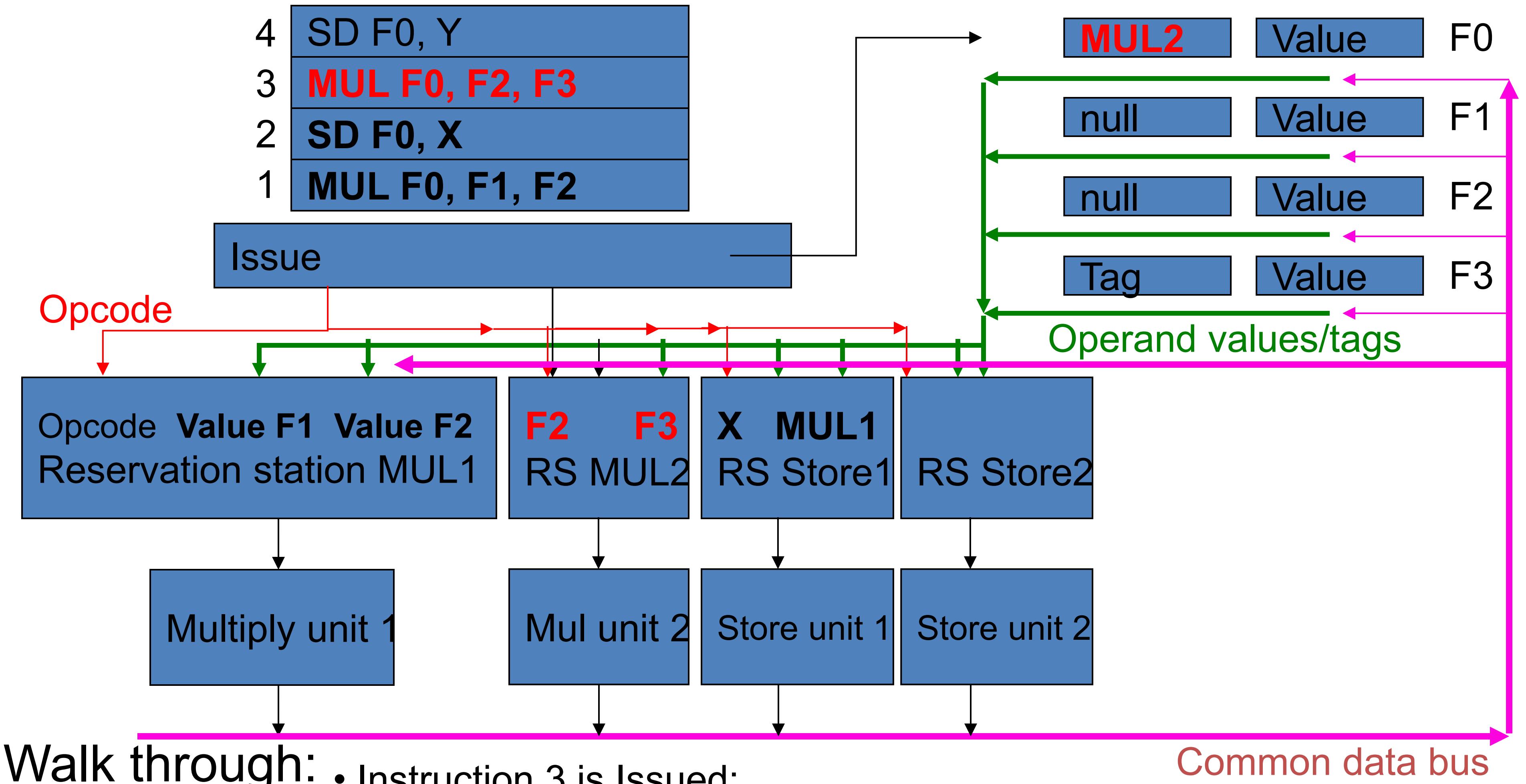
- Instruction 1 is Issued:
  - reservation station MUL1 is selected since it's free
  - tag of F1 is null so its value is routed to MUL1's operand 1
  - tag of F2 is null so its value is routed to MUL1's operand 2
  - tag of F0 is updated with id of MUL1, indicating that its value *will* come from MUL1

Tomasulo – **Walkthrough**



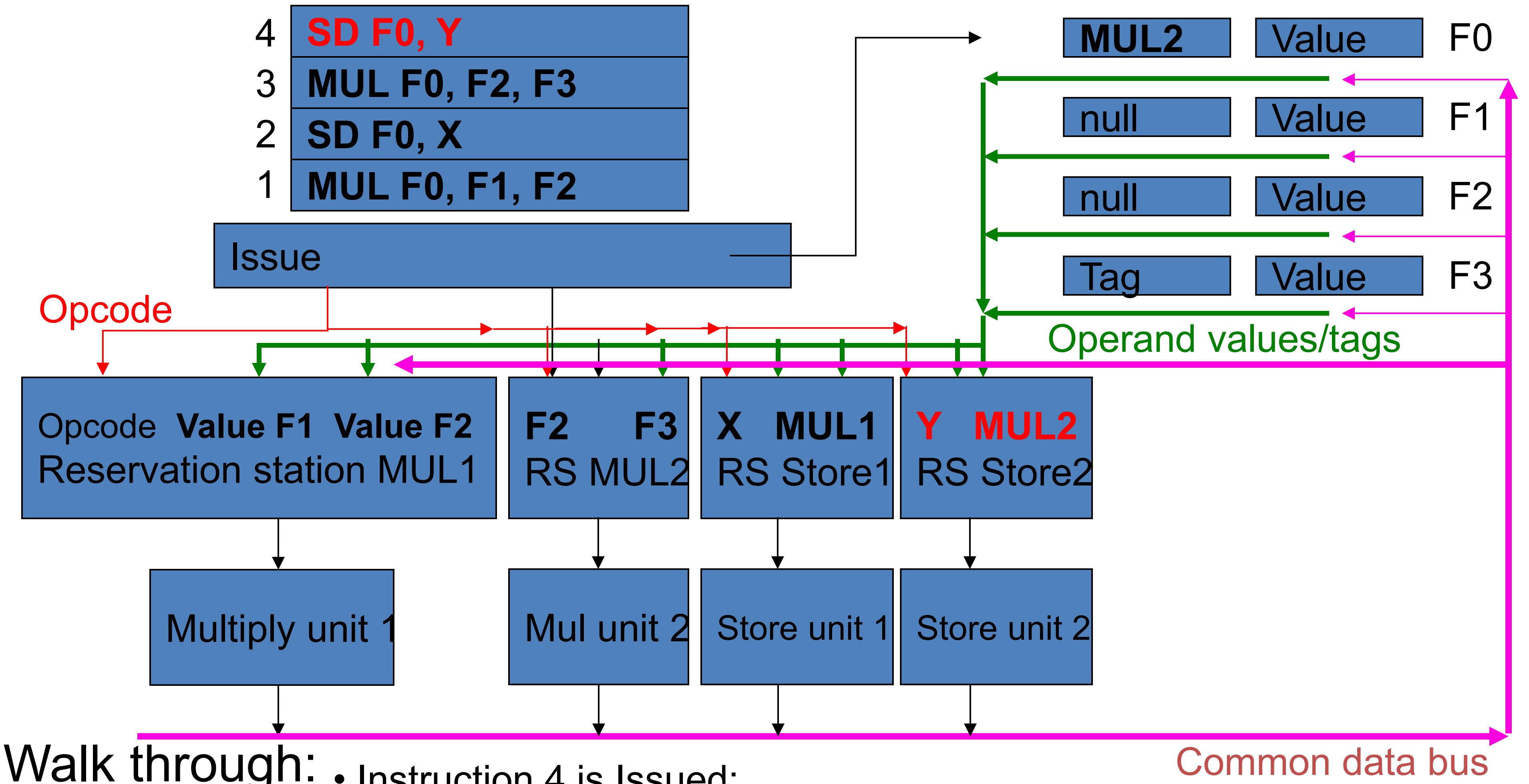
Walk through:
- Instruction 2 is Issued:
  - reservation station Store 1 is selected since it's free
  - tag of F0 is MUL1 so its *tag* is routed to Store 1's operand
  - address X is routed to Store 1

Tomasulo – **Walkthrough**



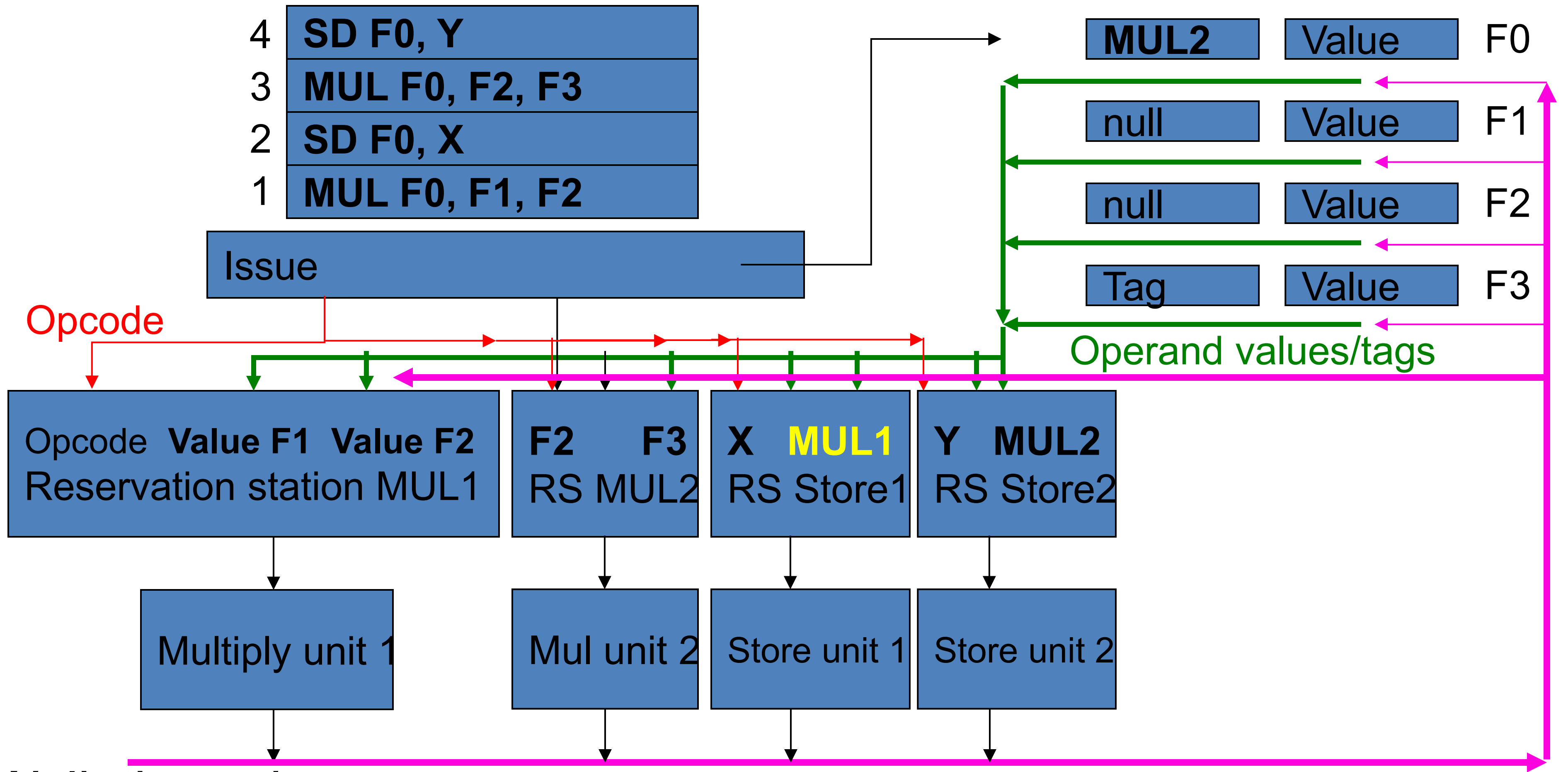
Walk through:
- Instruction 3 is Issued:
  - reservation station MUL2 is selected since it's free
  - tag of F2 is null so its value is routed to MUL2's operand 1
  - tag of F3 is null so its value is routed to MUL2's operand 2
  - tag of F0 is *overwritten* with id of **MUL2**, indicating that its value *will* come from MUL2

Tomasulo – **Walkthrough**



Walk through:
- Instruction 4 is Issued:
  - reservation station Store 2 is selected since it's free
  - tag of F0 is **MUL2** so its *tag* is routed to Store 2's operand
  - address Y is routed to Store 2

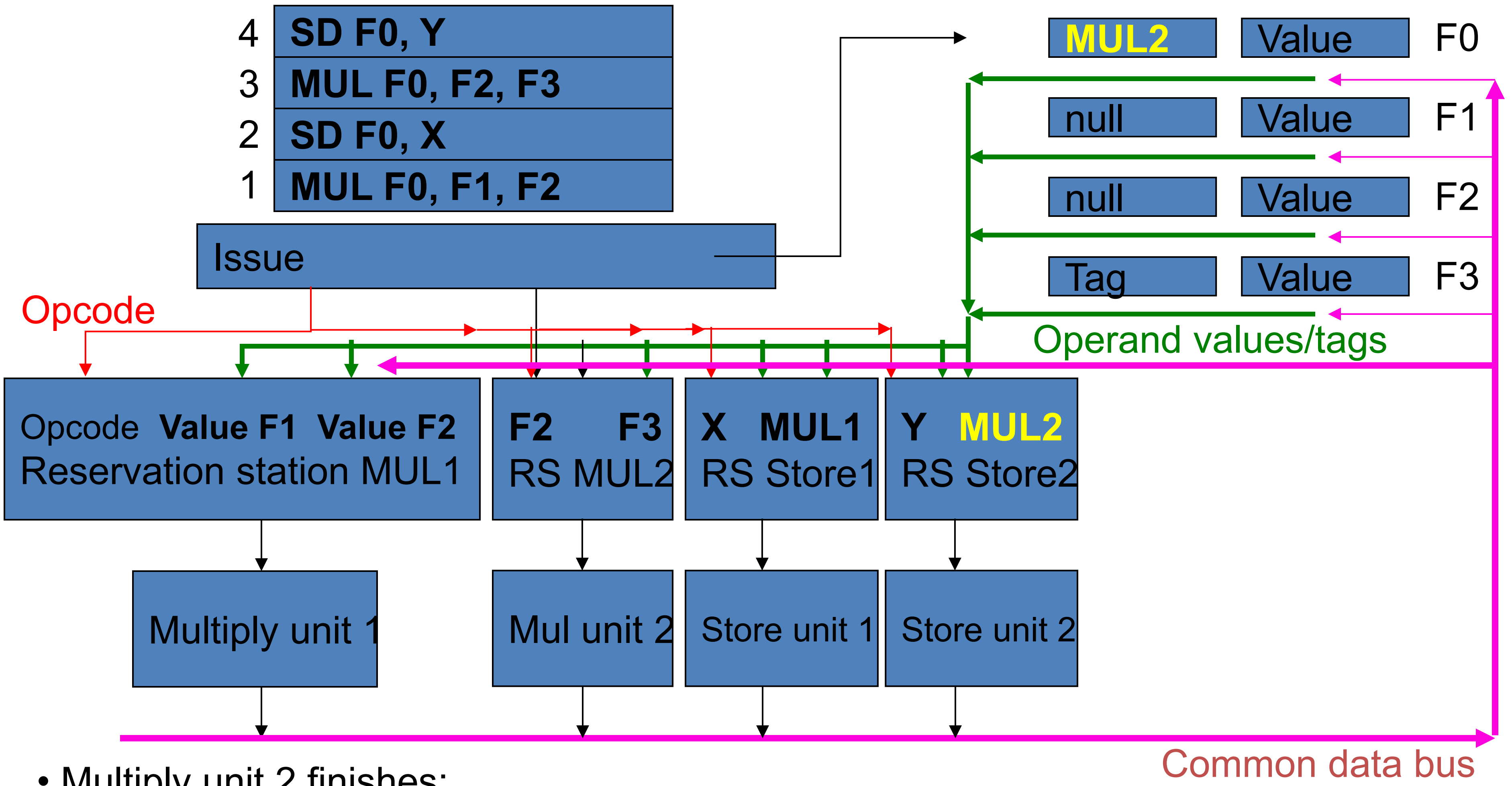
Tomasulo – **Walkthrough**



Walk through:

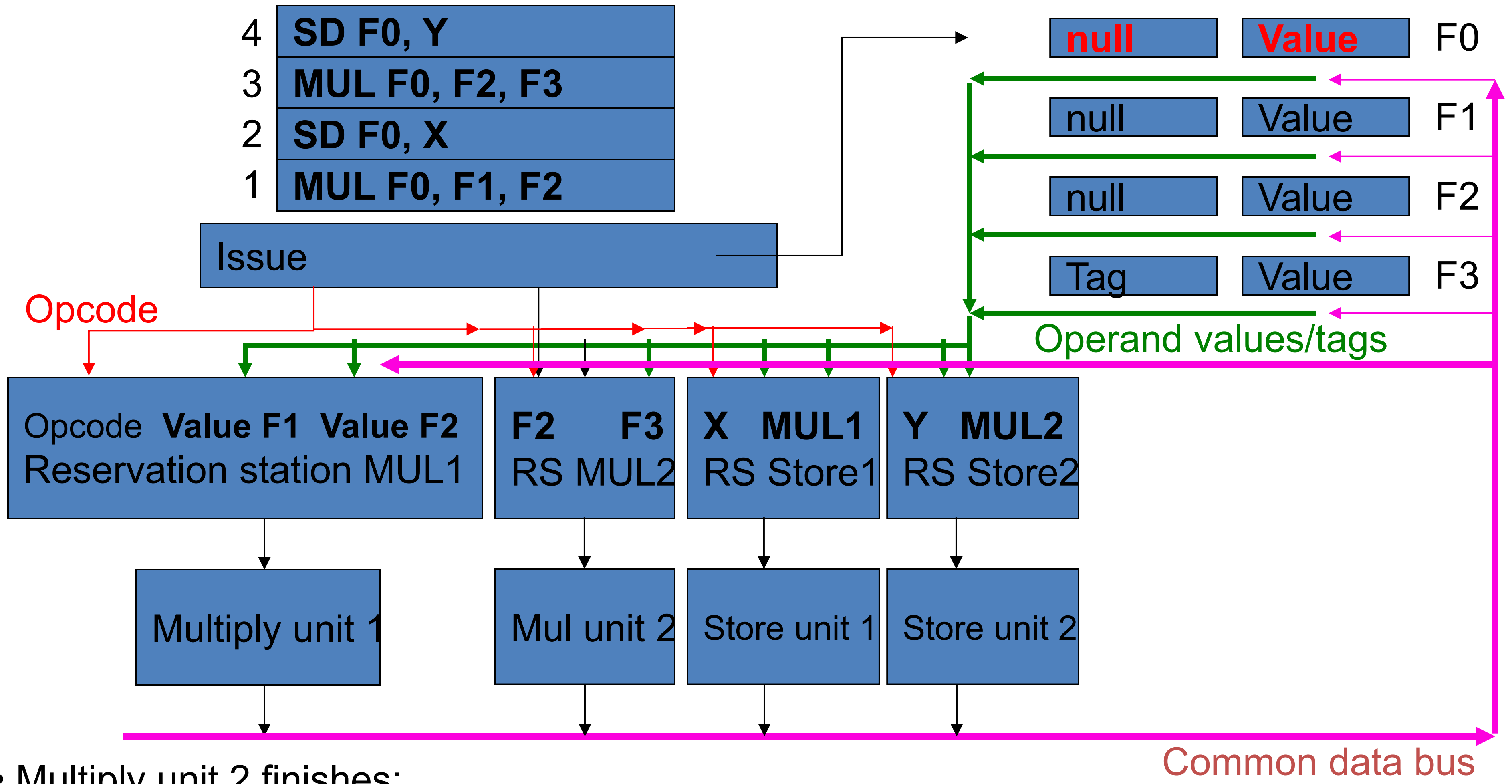
- Multiply unit 1 finishes:
  - It broadcasts its result on the Common Data Bus (CDB)
  - carrying the tag "MUL1"
  - Store 1 monitors the CDB, is waiting for a value with tag "MUL1"
  - Store 1 picks up the value and stores it to memory
  - (*Register F0 ignores this because it is waiting for a different tag*)

Tomasulo – **Walkthrough**



- Multiply unit 2 finishes:
  - It broadcasts its result on the Common Data Bus (CDB)
  - carrying the tag “MUL2”
  - Store 2 monitors the CDB, is waiting for a value with tag “MUL2”
  - Store 2 picks up the value and stores it to memory
  - Register F0 monitors CDB, sees “MUL2”, updates its value, sets F0’s tag to “null”

Tomasulo – **Walkthrough**



- Multiply unit 2 finishes:
  - It broadcasts its result on the Common Data Bus (CDB)
  - carrying the tag “MUL2”
  - Store 2 monitors the CDB, is waiting for a value with tag “MUL2”
  - Store 2 picks up the value and stores it to memory
  - Register F0 monitors CDB, sees “MUL2”, updates its value, sets F0’s tag to “null”

Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

   If reservation station free (no structural hazard),
   control issues instr & sends operands (renames registers).

2. Execute—operate on operands (EX)

   When both operands ready then execute;
   if not ready, watch Common Data Bus for result

3. Write result—finish execution (WB)

   Write on Common Data Bus to all awaiting units;
   mark reservation station available

Two buses:

- Normal data bus: data+destination ("go to" bus)
  – Used at Issue
- Common data bus: data+source ("come from" bus)
  – Used at WB
  – 64 bits of data + 4 bits of Functional Unit source address
  – Write if matches expected Functional Unit (produces result)
  – Does the broadcast

360/91 pipeline

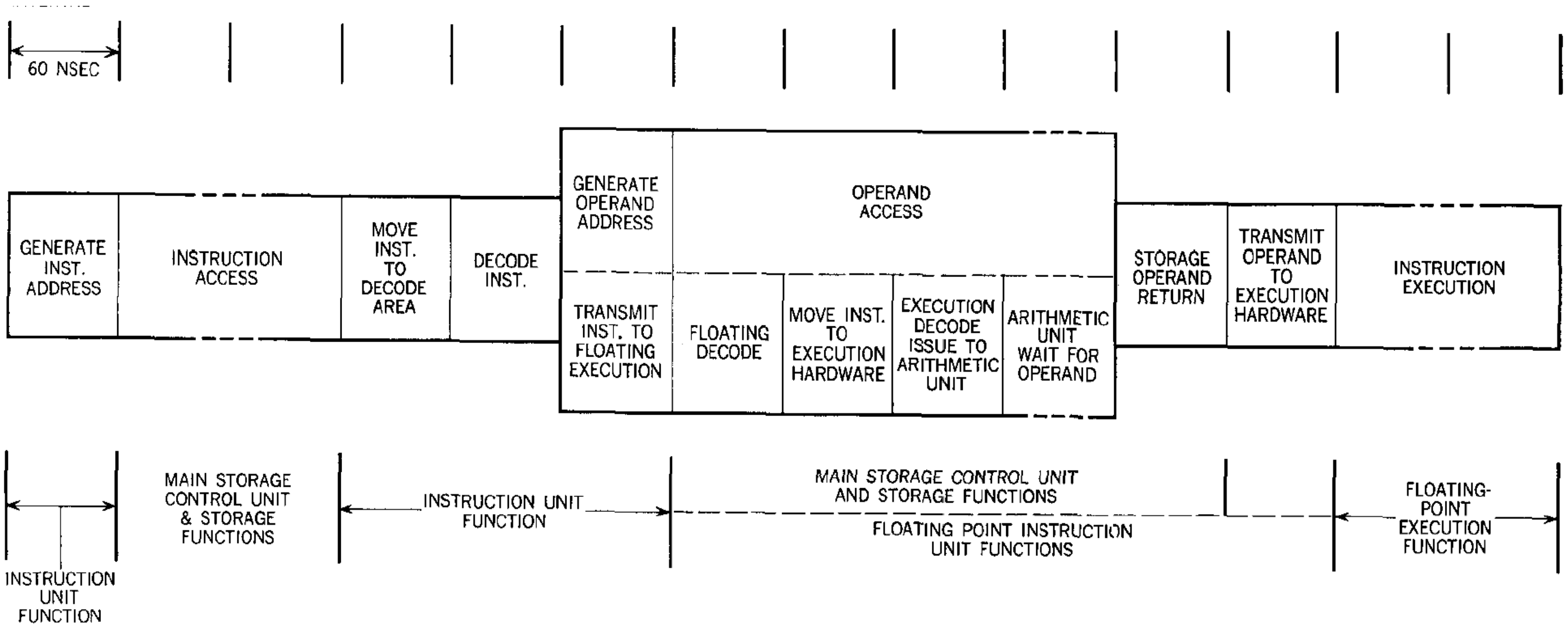

60 NSEC
GENERATE INST. ADDRESS
INSTRUCTION ACCESS
MOVE INST. TO DECODE AREA
DECODE INST.
GENERATE OPERAND ADDRESS
OPERAND ACCESS
TRANSMIT INST. TO FLOATING EXECUTION
FLOATING DECODE
MOVE INST. TO EXECUTION HARDWARE
EXECUTION DECODE ISSUE TO ARITHMETIC UNIT
ARITHMETIC UNIT WAIT FOR OPERAND
STORAGE OPERAND RETURN
TRANSMIT OPERAND TO EXECUTION HARDWARE
INSTRUCTION EXECUTION
INSTRUCTION UNIT FUNCTION
MAIN STORAGE CONTROL UNIT & STORAGE FUNCTIONS
INSTRUCTION UNIT FUNCTION
MAIN STORAGE CONTROL UNIT AND STORAGE FUNCTIONS
FLOATING POINT INSTRUCTION UNIT FUNCTIONS
FLOATING-POINT EXECUTION FUNCTION

Figure 3 CPU "assembly-line stations required to accommodate a typical floating-point storage-to-register instruction.

- 11-12 circuit levels per pipeline stage, of 5-6ns each
- CPU consists of three physical frames, each having dimensions 66" L X 15" D X 78" H

See: The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling,by D. W. Anderson, F. J. Sparacio, R. M. Tomasulo. IBM J. R&D (1967), http://www.research.ibm.com/journal/rd/111/ibmrd1101C.pdf

Tomasulo Drawbacks

- Complexity
  - delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620
  - Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
  - Each CDB must go to multiple functional units ⇒high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one!
    - Multiple CDBs ⇒ more FU logic for parallel assoc stores
- Non-precise interrupts!
  - We will address this later

Why can Tomasulo overlap iterations of loops?

- Register renaming
  - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).
- 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).
- Other perspective:
  - The CDB is doing forwarding, bypassing the registers
  - Builds the data flow dependency graph on the fly

Tomasulo Loop Example

```
Loop:
        LD          F0      0(R1)
        MULTD       F4      F0      F2
        SD          F4      0(R1)
        SUBI        R1      R1      #8
        BNEZ  R1    Loop
```

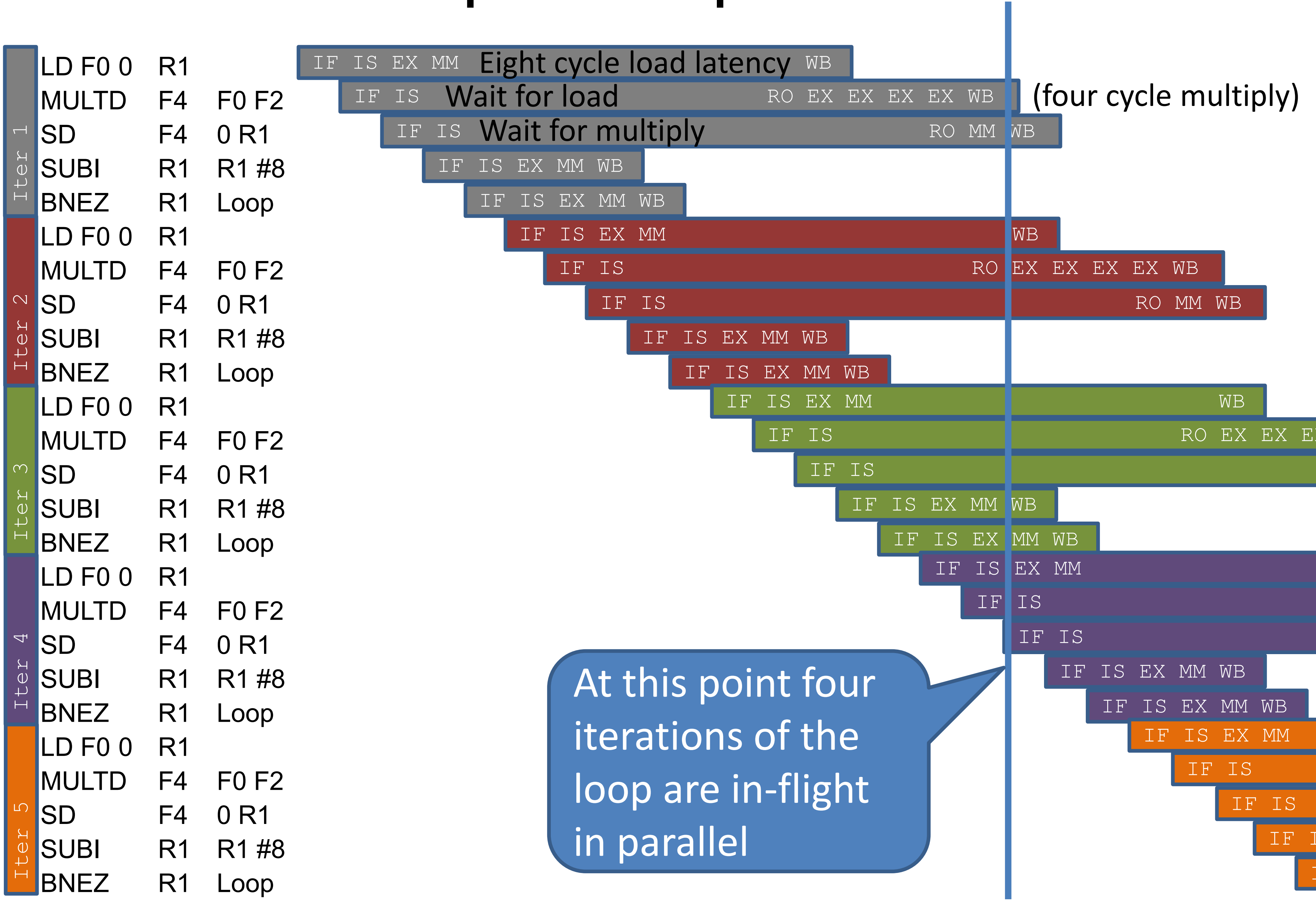
- Assume floating-point multiply takes 4 clocks
- Suppose loads take 8 clocks (L1 cache miss)

  ((Actually each L1 cache miss would load a cache line of several words, and prefetching might reduce latency of next fetch))

  ((example counts R1 down to 0 in order to simplify the loop for the sake of the example))
- Assume that integer instructions don’t use the CDB
- Assume SD doesn’t use the CDB

Tomasulo Loop Example

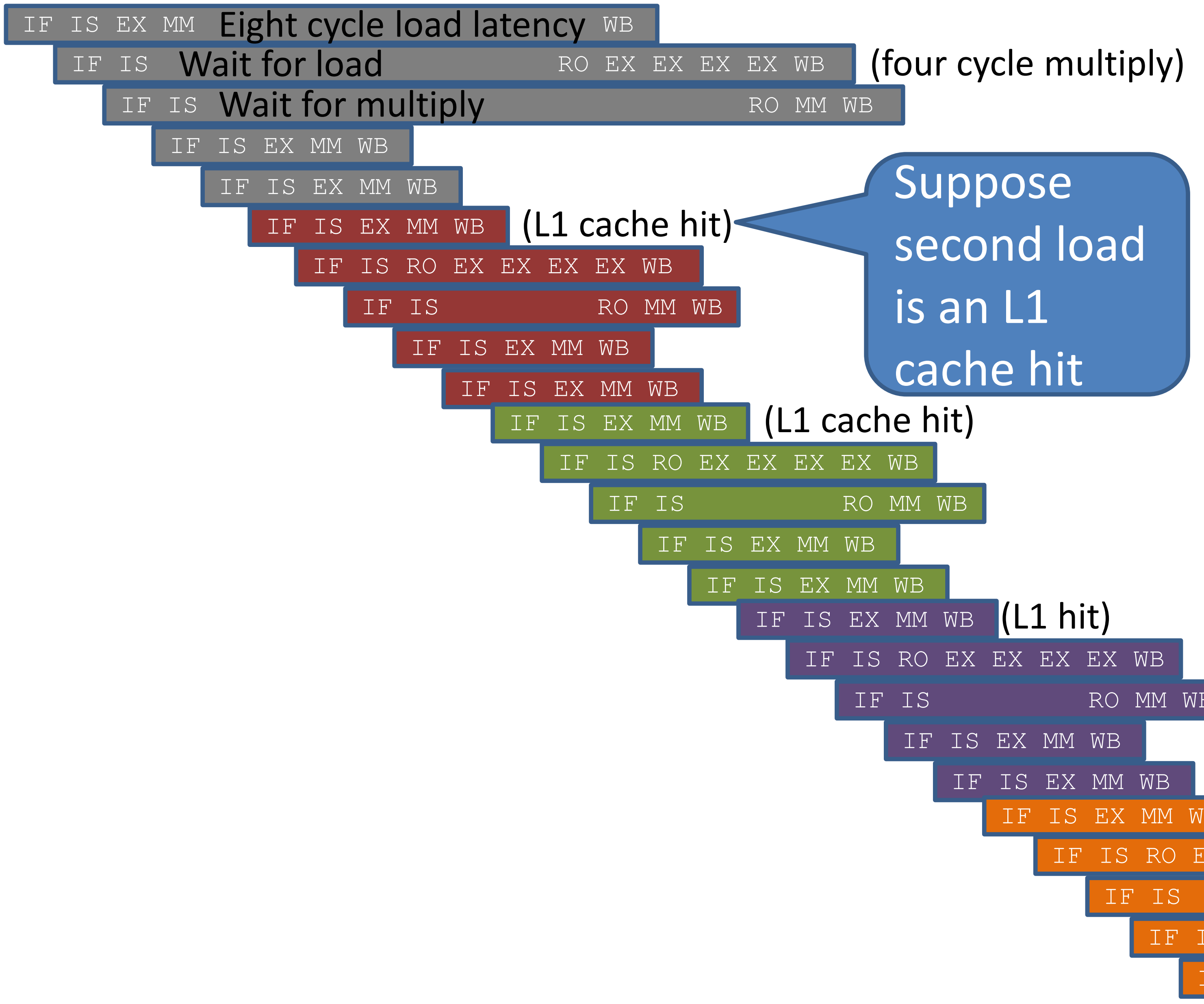
```
LD    F0 0  R1
MULTD F4    F0 F2
SD    F4    0 R1
SUBI  R1    R1 #8
BNEZ  R1    Loop
LD    F0 0  R1
MULTD F4    F0 F2
SD    F4    0 R1
SUBI  R1    R1 #8
BNEZ  R1    Loop
LD    F0 0  R1
MULTD F4    F0 F2
SD    F4    0 R1
SUBI  R1    R1 #8
BNEZ  R1    Loop
LD    F0 0  R1
MULTD F4    F0 F2
SD    F4    0 R1
SUBI  R1    R1 #8
BNEZ  R1    Loop
LD    F0 0  R1
MULTD F4    F0 F2
SD    F4    0 R1
SUBI  R1    R1 #8
BNEZ  R1    Loop
```



At this point four iterations of the loop are in-flight in parallel

Tomasulo Loop Example

Iter 1	LD F0 0	R1	
	MULTD	F4	F0 F2
	SD	F4	0 R1
	SUBI	R1	R1 #8
	BNEZ	R1	Loop
Iter 2	LD F0 0	R1	
	MULTD	F4	F0 F2
	SD	F4	0 R1
	SUBI	R1	R1 #8
	BNEZ	R1	Loop
Iter 3	LD F0 0	R1	
	MULTD	F4	F0 F2
	SD	F4	0 R1
	SUBI	R1	R1 #8
	BNEZ	R1	Loop
Iter 4	LD F0 0	R1	
	MULTD	F4	F0 F2
	SD	F4	0 R1
	SUBI	R1	R1 #8
	BNEZ	R1	Loop
Iter 5	LD F0 0	R1	
	MULTD	F4	F0 F2
	SD	F4	0 R1
	SUBI	R1	R1 #8
	BNEZ	R1	Loop



Summary: Tomasulo

- RAW, WAR and WAW hazards
- Tomasulo overcomes WAR and WAW hazards by dynamically allocating operands to reservation stations at issue time
  - Register renaming, seen more explicitly in later designs
- Tomasulo's CDB is a kind of "forwarding" path – that routes operands from completing FUs to where they are needed
  - In multi-issue processors this gets a lot more complicated!
- Tomasulo's scheme relies on associative tag matching
  - Later designs assign physical registers explicitly to avoid this
- Tomasulo's scheme enables multiple FUs to operate in parallel
  - Even across loop iterations

Student questions:

Consider this example:

1- MUL F0, F1, F2

2- MUL F0, F0, F3

3- SD F0, X

Let iX denote instruction X (example: i1 denotes the first instruction)

If I understood correctly:

- a- i1 will check that dependencies F1 and F2 are free, reserves a MUL1 station, and tags F0 with MUL1
- b- i2 finds out F0 is awaiting result. It reserves a MUL2 station with operands MUL1 (tag of F0) and F3, **then** tags F0 with MUL2, and awaits a MUL1 tag check from CDB
- c- i3 reserves a Store1 station, with operands MUL2 and X, and awaits CDB MUL2 tag
- d- MUL1 station finishes executing i1. MUL1 tag propagates through CDB and triggers station MUL2
- e- MUL2 station can now execute and finishes executing i2. MUL2 tag propagates through CDB and triggers station Store1. It also writes over F0 value *(and sets its tag to NULL?)*
- f- Store1 executes and finishes

Another example:

1- MUL F0, F1, F2

2- MUL F0, F0, F3

3- ADD F3, F1, F2

4- SD F0, X

“If F3 was just wired to i2 MUL2 station, then ADD would have changed the value and that's a WAR hazard”?

- F3 is read for i2 at i2's *issue* time. Whatever is there (value or tag) is *copied* to the ADD reservation station.
- So when i3 overwrites F3, it's fine because the MUL2 RS is already holding the right thing for its F3 operand.

Student questions:

Yes basically the natural way to extend Tomasulo to support >1 instruction per cycle is to build multiple CDBs.

Which means that every unit that monitors the CDB now has to monitor (and do a tag comparison on every CDB transaction) on all the CDBs.

We don't have time to dive into how to do better than this but it's possible. A starting point for understanding how is https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp16/cse502/res/R10k.pdf .

If you really want to dive deep, here's a link into the documentation for BOOM, an open-source multi-issue out-of-order processor design: https://docs.boom-core.org/en/latest/sections/reorder-buffer.html#the-commit-stage .

From there you can find the code that does it. See: https://github.com/riscv-boom/riscv-boom/blob/master/src/main/scala/exu/rob.scala eg from line 401.

Incidentally we discussed what happens when multiple instructions complete at the same time - we get contention for the CDB(s). We might wonder what the best policy for selecting which one should go first is. This paper tries to provide an answer for what the optimal thing to do is: Lin-and-Tian.pdf (washington.edu)

By the way:

Bio: https://en.wikipedia.org/wiki/Robert_Tomasulo

Paper: R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," in IBM Journal of Research and Development, vol. 11, no. 1, pp. 25-33, Jan. 1967, doi: 10.1147/rd.111.0025.
https://www.cs.virginia.edu/~evans/greatworks/tomasulo.pdf

Talk: http://leccap.engin.umich.edu/leccap/player/r/pvSbKs

Student question – pathway to real stuff:

Q:

- **What would I need beyond this course to go about designing hardware? I've seen some work by professors to do with FPGA, but its not super clear to me what that takes.**
- **Say you had some algorithm that felt could be implemented with a digital circuit, what would you do?**

We have other students in the class with excellent experience, both at university and in industry, to address your question.

- The ACA course is primarily about designing processors - for which you would probably use a Register Transfer Level (RTL, Register-transfer level - Wikipedia) programming model. Popular languages for this include Verilog, VHDL and Chisel (a DSL embedded in Scala). Basically you write code that describes the wiring of a digital circuit. You can then run it on your PC in simulation (see Verilator for example), or compile to an FPGA configuration, or (via a more involved route) to a VLSI layout.
- What architects actually do is use a hierarchy of models - for example using gem5 to simulate the microarchitecture at the block-diagram level (this is basically the level at which this course operates). You might then have a "cycle accurate" model, for example built using SystemC (though not everyone does this). Then you would elaborate this into an RTL design - where you have to describe a digital circuit that actually does the work that needs to be done in each clock cycle.
- RTLs like Verilog have further subtleties - you can write "behavioural" Verilog, that works in simulation, but to generate actual hardware you need to write in "synthesisable" Verilog.
- Particular companies typically have a rich ecosystem of modelling and verification tools, and there is also a large ecosystem of verification and testing tools.
- This all sounds a bit abstract. You can go and read the source code (in Chisel) for a non-trivial CPU architecture here: GitHub - riscv-boom/riscv-boom: SonicBOOM: The Berkeley Out-of-Order Machine (https://github.com/riscv-boom/riscv-boom) ; see the documentation here https://boom-core.org/ (you can also find source code for much simpler CPU designs elsewhere of course).
- So for example this week's lecture on Tomasulo's algorithm introduces out-of-order architecture. Check it out for real here: https://docs.boom-core.org/en/latest/sections/reorder-buffer.html . Go read the code for the ROB here: https://github.com/riscv-boom/riscv-boom/blob/master/src/main/scala/v4/exu/rob.scala (the ROB is actually not covered until the next lecture, on speculative execution).

BUT: you also ask about accelerator architecture - ie a digital circuit to compute some specific thing, in contrast to a general-purpose CPU.

- For this there are tools like look more like software - High-Level Synthesis (HLS) (https://en.wikipedia.org/wiki/High-level_synthesis). Although there have been many high-level synthesis tools, these days people usually mean you start from C code and add directives to specify how the C code should be mapped into hardware. There are of course also high-level tools that start from a more domain-specific representation, such as a deep neural network.