332
Advanced Computer Architecture
Chapter 3

Dynamic scheduling, out-of-order execution, register renaming and speculative execution

January 2018

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 at
http://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture.html

Advantages of Dynamic Scheduling

- Handles cases when dependences unknown at compile time
  - (e.g., because they may involve a memory reference)
- It simplifies the compiler
- Allows code that compiled for one pipeline to run efficiently on a different pipeline
- Hardware speculation, a technique with significant performance advantages, builds on dynamic scheduling

HW Schemes: Instruction Parallelism

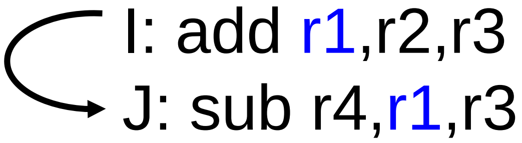
- Key idea: Allow instructions behind stall to proceed

```
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

- Instr$_J$ is data dependent on Instr$_I$
  Instr$_J$ tries to read operand before Instr$_I$ writes it



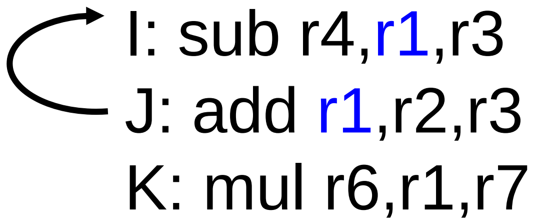
```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- 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 #1: Anti-dependence

- 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:



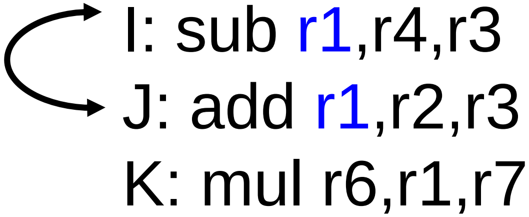
```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

  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

Name Dependence #2: Output dependence

- Instr$_J$ writes operand ***before*** Instr$_I$ writes it.



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

- 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

- 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

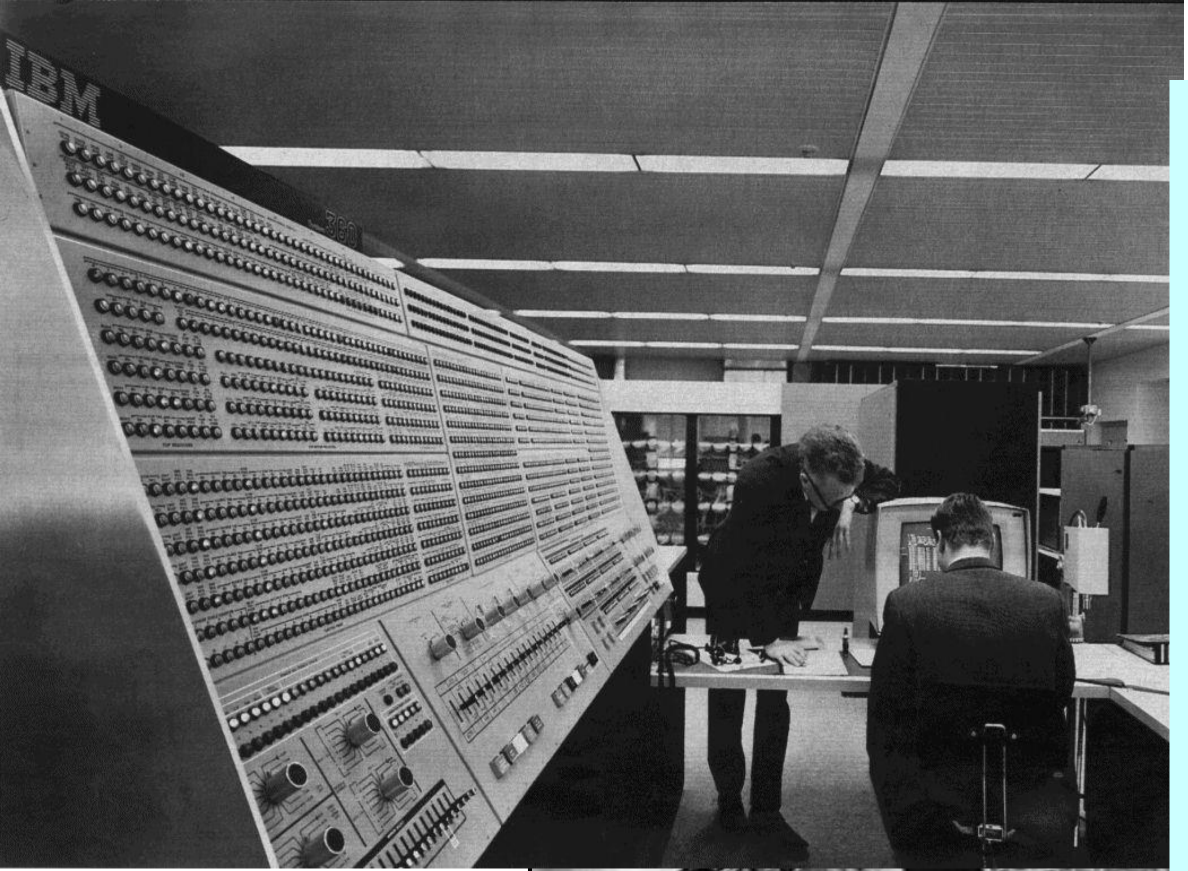
Instructions are *issued* in-order

But may stall at the Read Operands stage while others execute

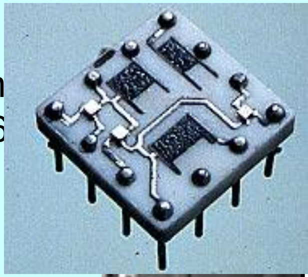
Tomasulo’s Algorithm

- For IBM 360/91 (before caches!)
- Goal: High Performance without special compilers
- Small number of floating point registers (4 in 360) prevented interesting 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

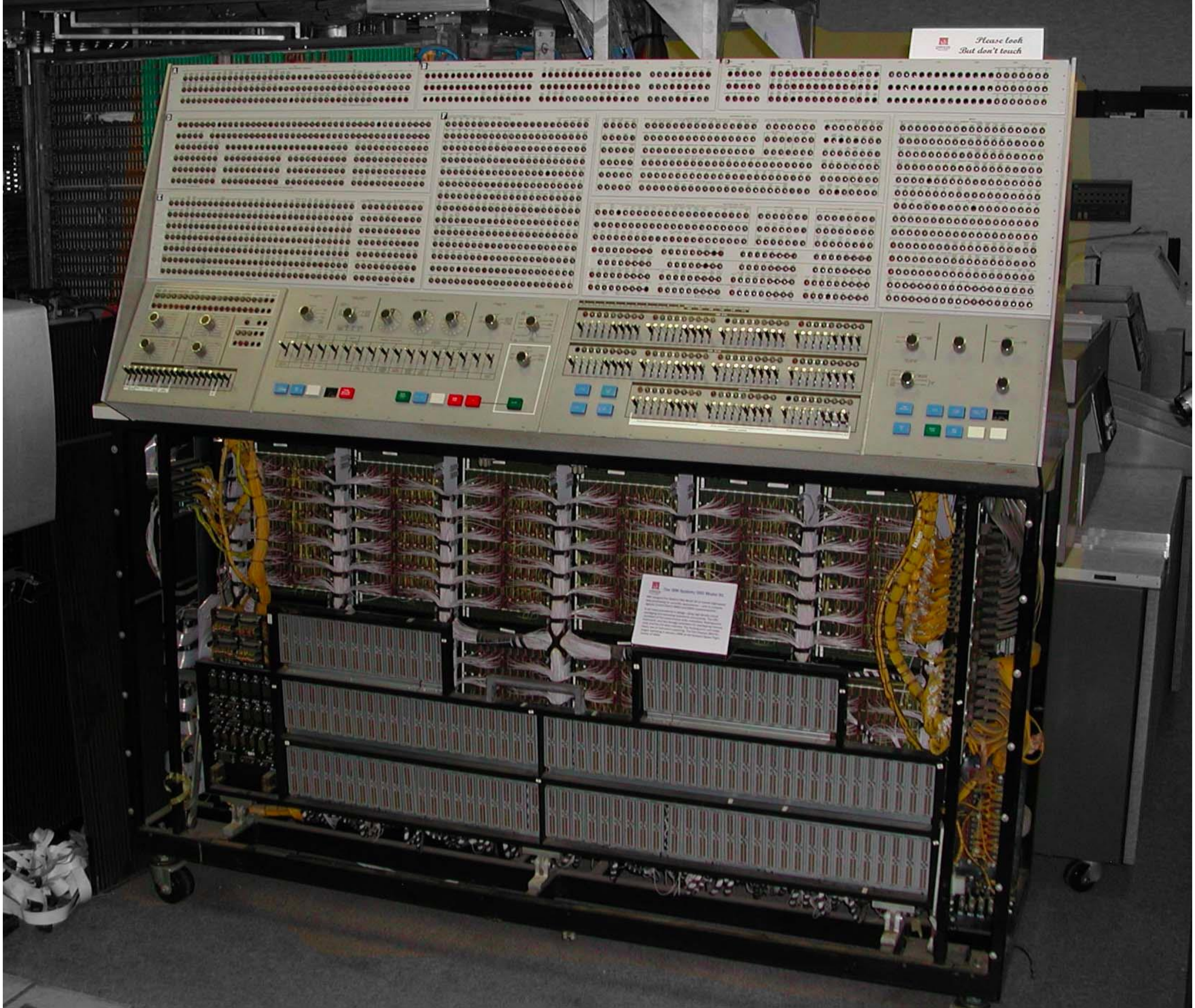


NASA Center for Computational Sciences

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

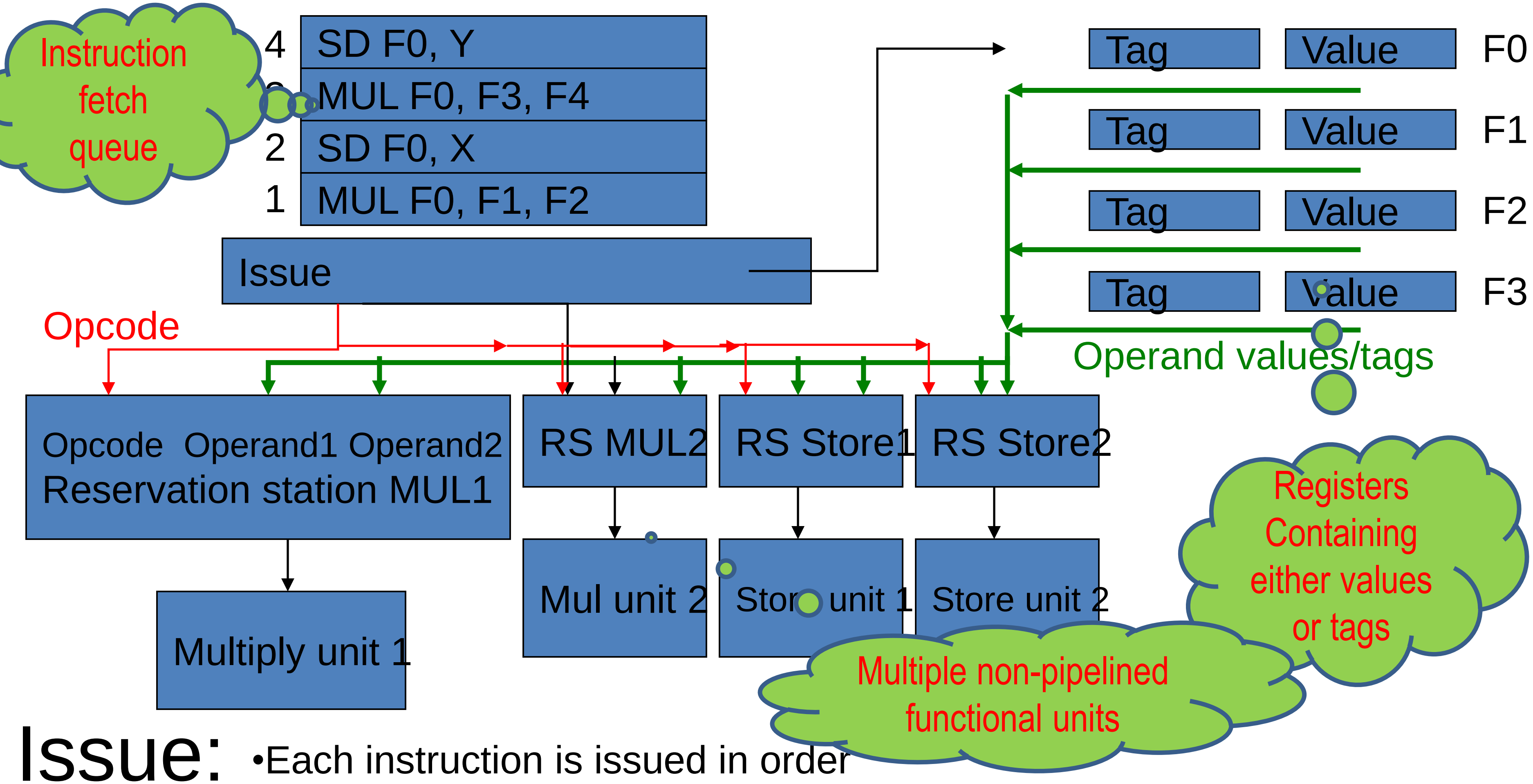
Source: http://www.columbia.edu/acis/history/36091.html

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



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

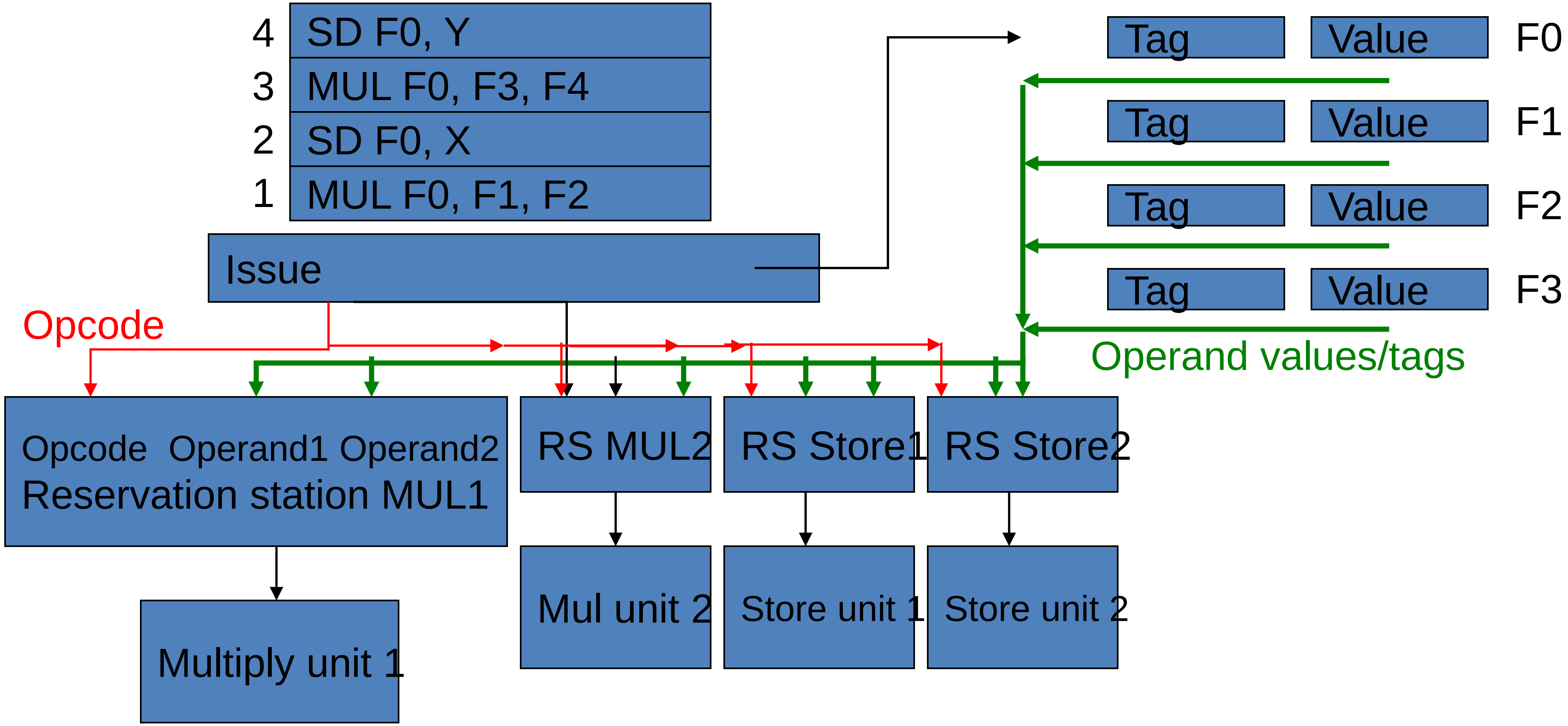
Tomasulo – closer look at instruction processing



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

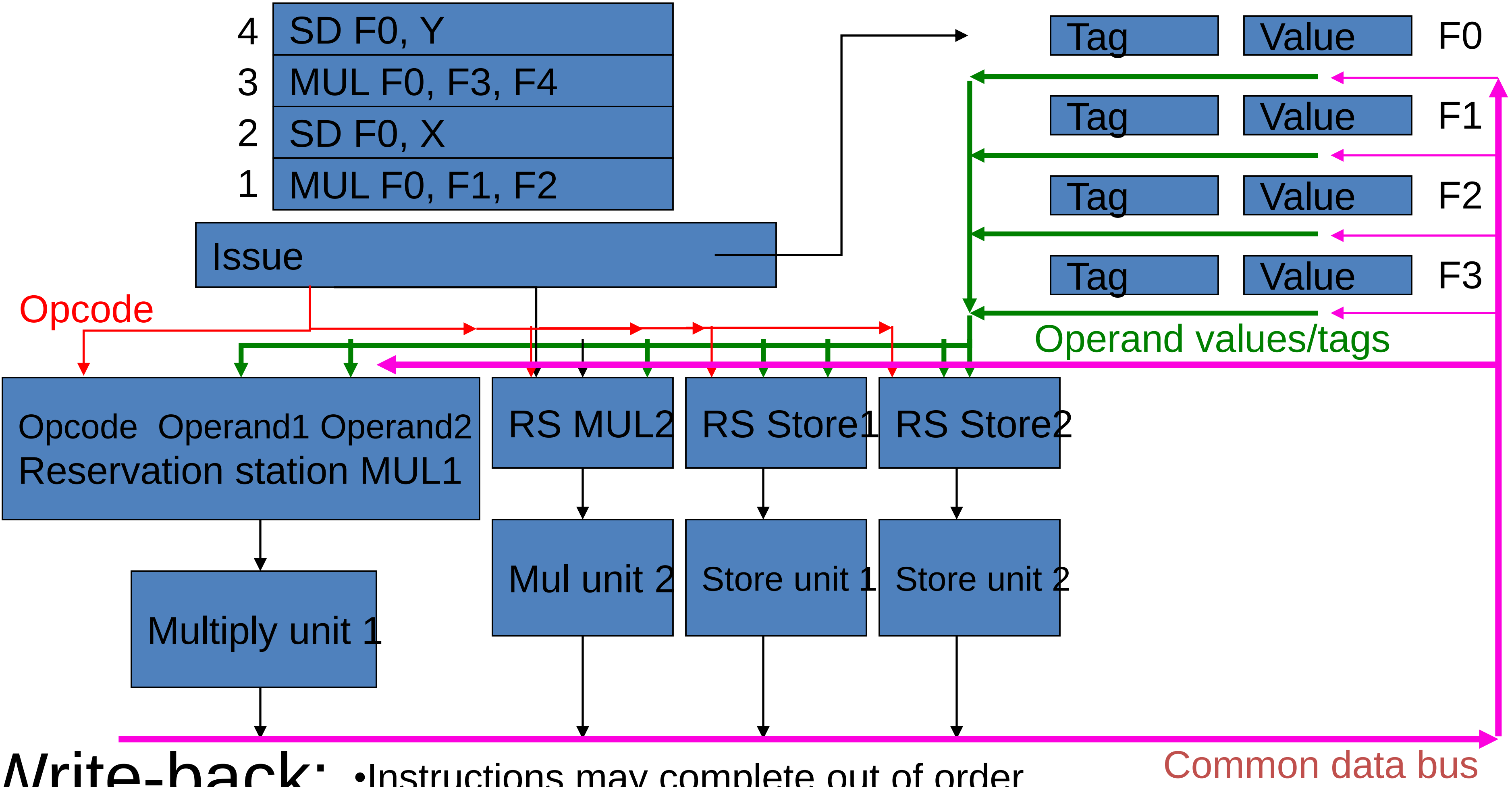
Tomasulo – closer look at instruction processing



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

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

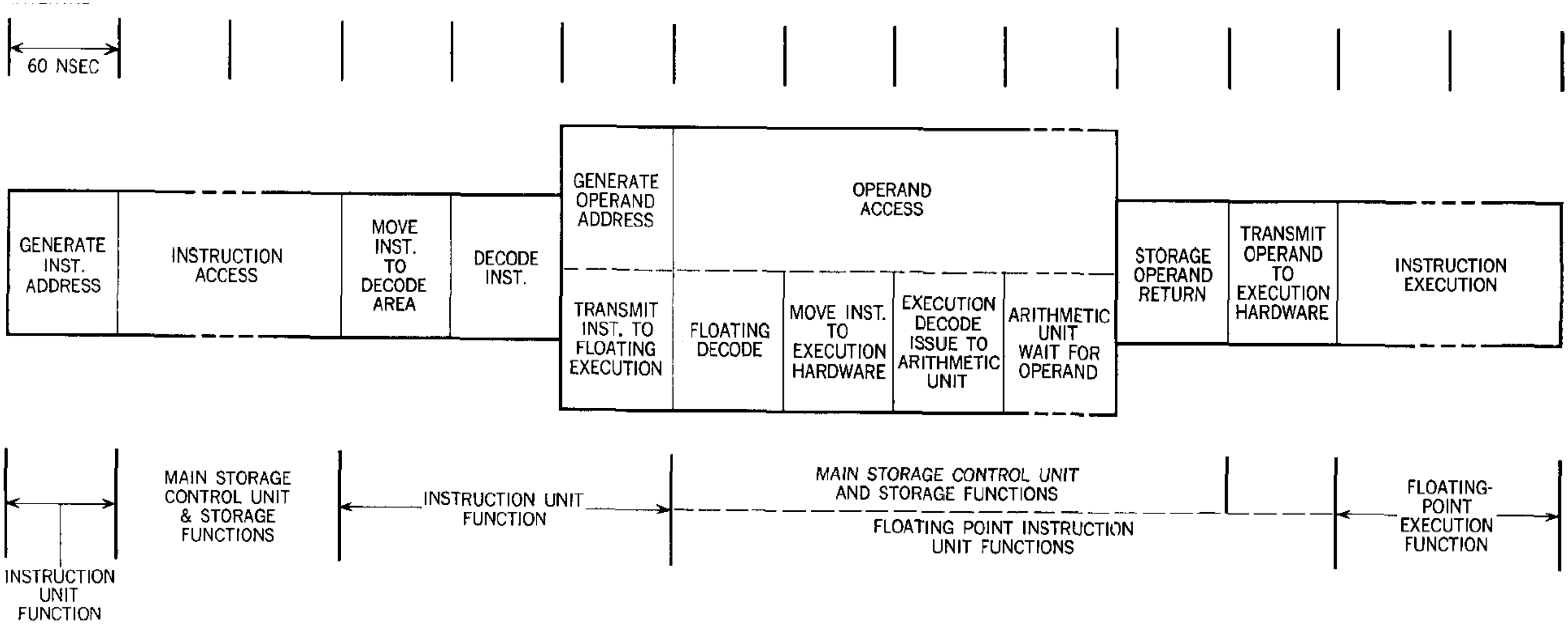

60 NSEC
GENERATE INST. ADDRESS
INSTRUCTION ACCESS
MOVE INST. TO DECODE AREA
DECODE INST.
GENERATE OPERAND ADDRESS
OPERAND ACCESS
TRANSMIT INST. TO FLOATING EXECUTION
FLOATING DECODE
MOVE INST. TO EXECUTION HARDWARE
EXECUTION DECODE ISSUE TO ARITHMETIC UNIT
ARITHMETIC UNIT WAIT FOR OPERAND
STORAGE OPERAND 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

Tomasulo Loop Example

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

- Assume Multiply takes 4 clocks
- Assume 1st load takes 8 clocks
  (L1 cache miss), 2nd load takes 1 clock (hit)
- To be clear, will show clocks for SUBI, BNEZ
  - Reality: integer instructions ahead of Fl. Pt. Instructions
- Show 2 iterations

Loop Example

Instruction status:

						Exec	Write
ITER	Instruction		j	k	Issue	Comp	Result
1	LD	F0	0	R1			
1	MULTD	F4	F0	F2			
1	SD	F4	0	R1			
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

Iter-ation Count

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Added Store Buffers

Reservation Stations:

					S1	S2	RS
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

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

Instruction Loop

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
0	80	Fu									

Value of Register used for address, iteration control

Reservation Station Components

Op:Operation to perform in the unit (e.g., + or –)

Vj, Vk: Value of Source operands

- Store buffers has V field, result to be stored

Qj, Qk: Reservation stations producing source registers (value to be written)

- Note: Qj,Qk=0 => ready
- Store buffers only have Qi for RS producing result

Busy: Indicates reservation station or FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Loop Example Cycle 1

Instruction status:

						Exec	Write
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1		

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
1	**80**	*Fu*	Load1								

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

					S1	S2	RS
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
2	80	Fu	Load1		Mult1						

Loop Example Cycle 3

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
3	**80**	*Fu*	Load1		Mult1						

- Implicit renaming sets up data flow graph

Loop Example Cycle 4

Instruction status:

						Exec	*Write*
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
4	**80**	*Fu*	Load1		Mult1						

- Dispatching SUBI Instruction (not in FP queue)

Loop Example Cycle 5

Instruction status:

						Exec	*Write*
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
5	**72**	*Fu*	Load1		Mult1						

- And, BNEZ instruction (not in FP queue)

Loop Example Cycle 6

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
6	**72**	*Fu*	Load2		Mult1						

- Notice that F0 never sees Load from location 80

Loop Example Cycle 7

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
7	**72**	*Fu*	Load2		Mult2						

- Register file completely detached from computation
- First and Second iteration completely overlapped

Loop Example Cycle 8

Instruction status:

						Exec	*Write*
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	**72**	*Fu*	Load2		Mult2						

Loop Example Cycle 9

Instruction status:

						Exec	Write
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
9	**72**	*Fu*	Load2		Mult2						

- Load1 completing: who is waiting?
- Note: Dispatching SUBI

Loop Example Cycle 10

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	Multd	M[80]	R(F2)		
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	**64**	*Fu*	Load2		Mult2						

- Load2 completing: who is waiting?
- Note: Dispatching BNEZ

Loop Example Cycle 11

Instruction status:

						Exec	*Write*
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*	
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*	
	Add1	No						
	Add2	No						
	Add3	No						
3	Mult1	Yes	Multd	M[80]	R(F2)			
4	Mult2	Yes	Multd	M[72]	R(F2)			

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	**64**	*Fu*	Load3		Mult2						

- Next load in sequence

Loop Example Cycle 12

Instruction status:

						Exec	Write
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	Multd	M[80]	R(F2)		
3	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
12	**64**	*Fu*	Load3		Mult2						

- Why not issue third multiply?

Loop Example Cycle 13

Instruction status:

						Exec	*Write*
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	Multd	M[80]	R(F2)		
2	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
13	**64**	*Fu*	Load3		Mult2						

- Why not issue third store?

Loop Example Cycle 14

Instruction status:

						Exec	*Write*
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	Multd	M[80]	R(F2)		
1	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
14	**64**	*Fu*	Load3		Mult2						

- Mult1 completing. Who is waiting?

Loop Example Cycle 15

Instruction status:

						Exec	Write
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	**64**	*Fu*	Load3		Mult2						

- Mult2 completing. Who is waiting?

Loop Example Cycle 16

Instruction status:

						Exec	Write
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	No		

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 17

Instruction status:

						Exec	Write
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
17	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 18

Instruction status:

						Exec	Write
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
18	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 19

Instruction status:

						Exec	*Write*
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	19
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8	19	

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	No		
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

					S1	*S2*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
19	**56**	*Fu*	Load3		Mult1						

Loop Example Cycle 20

Instruction status:

						Exec	Write
ITER	Instruction		*j*	*k*	*Issue*	*Comp*	*Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	19
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8	19	20

	Busy	*Addr*	*Fu*
Load1	Yes	56	
Load2	No		
Load3	Yes	64	
Store1	No		
Store2	No		
Store3	Yes	64	Mult1

Reservation Stations:

					S1	*S2*	*RS*	
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*	
	Add1	No						
	Add2	No						
	Add3	No						
	Mult1	Yes	Multd		R(F2)	Load3		
	Mult2	No						

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
20	**56**	*Fu*	Load1		Mult1						

- Once again: In-order issue, out-of-order execution and out-of-order completion.

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 that we saw in the scoreboard.

- Other perspective:
  - The CDB is doing forwarding, bypassing the registers
  - Builds the data flow dependency graph on the fly

What about Precise Interrupts?

- Tomasulo had:

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

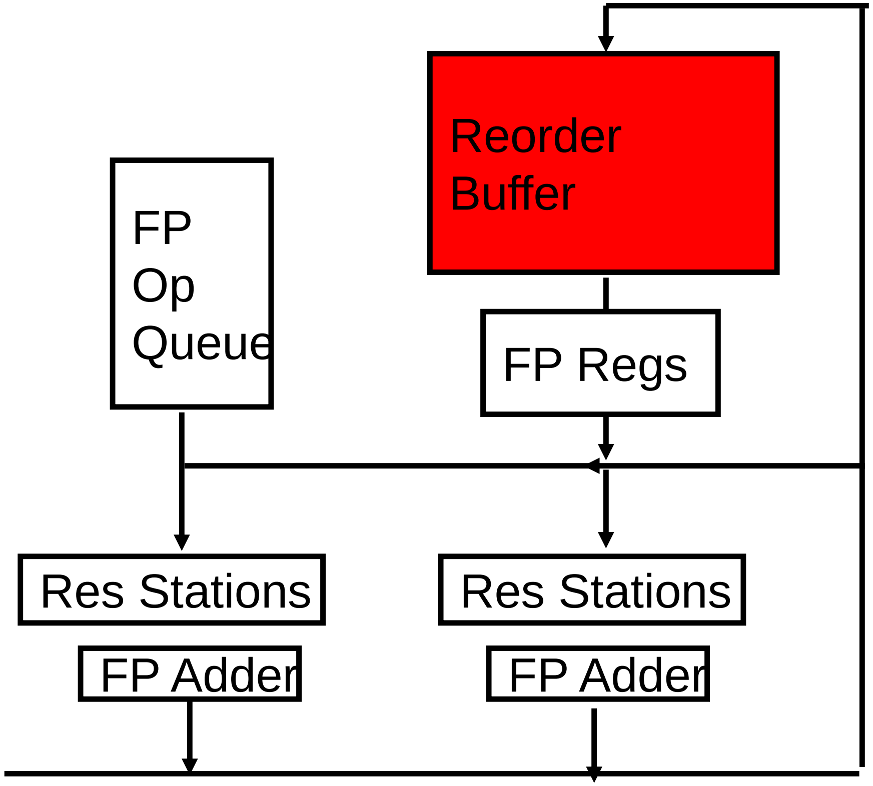
- Need to “fix” the out-of-order completion aspect so that we can find precise breakpoint in instruction stream

- Actually we have the same issue with branch speculation…

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

HW support for precise interrupts

- Need HW buffer for results of uncommitted instructions: *reorder buffer*
  - 3 fields: instr, destination, value
  - Use reorder buffer number instead of reservation station when execution completes
  - Supplies operands between execution complete & commit
  - (Reorder buffer can be operand source => more registers like RS)
  - Instructions commit
  - Once instruction commits,
result is put into register
  - As a result, easy to undo speculated instructions
on mispredicted branches
or exceptions



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")

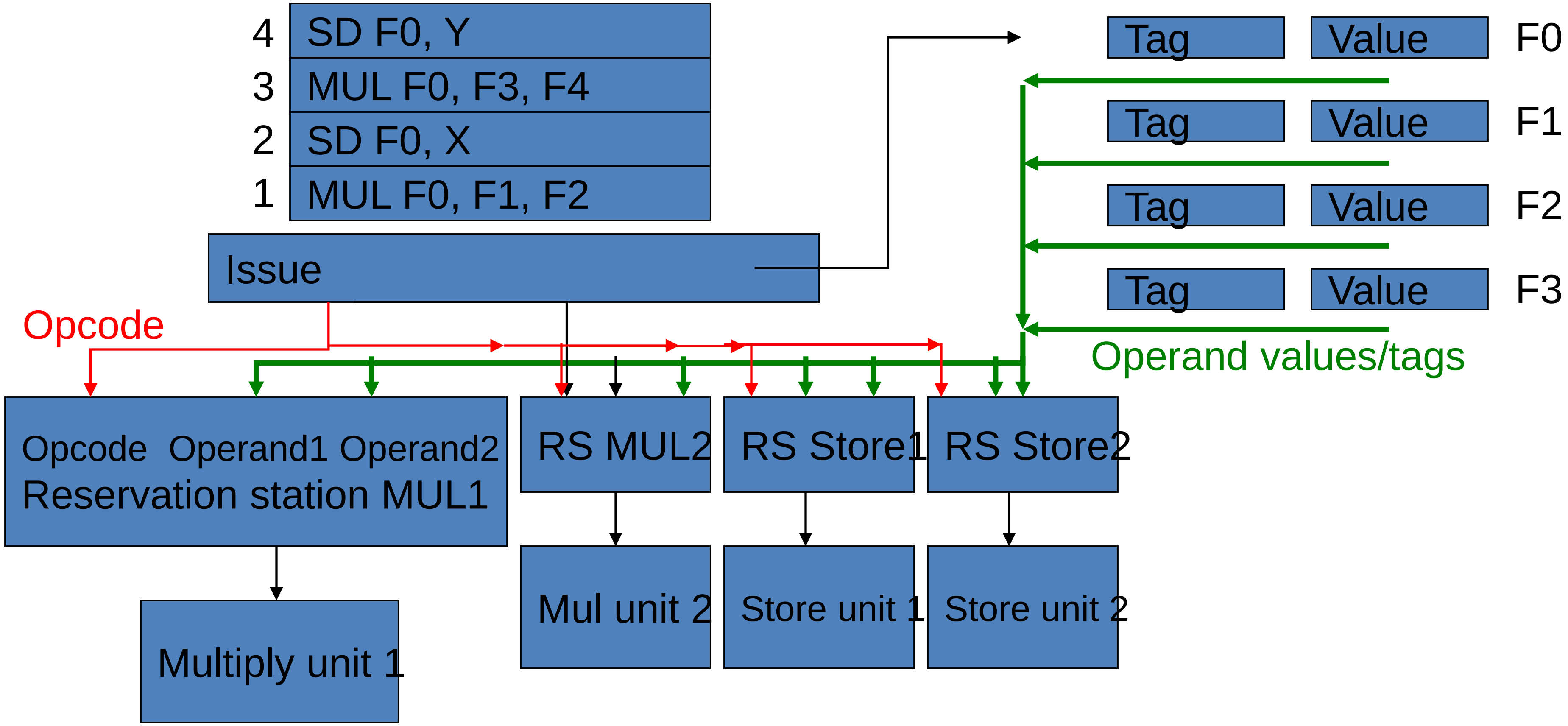
3. Write result—finish execution (WB)

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

4. Commit—update register with reorder result

   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")

Tomasulo without Re-order Buffer

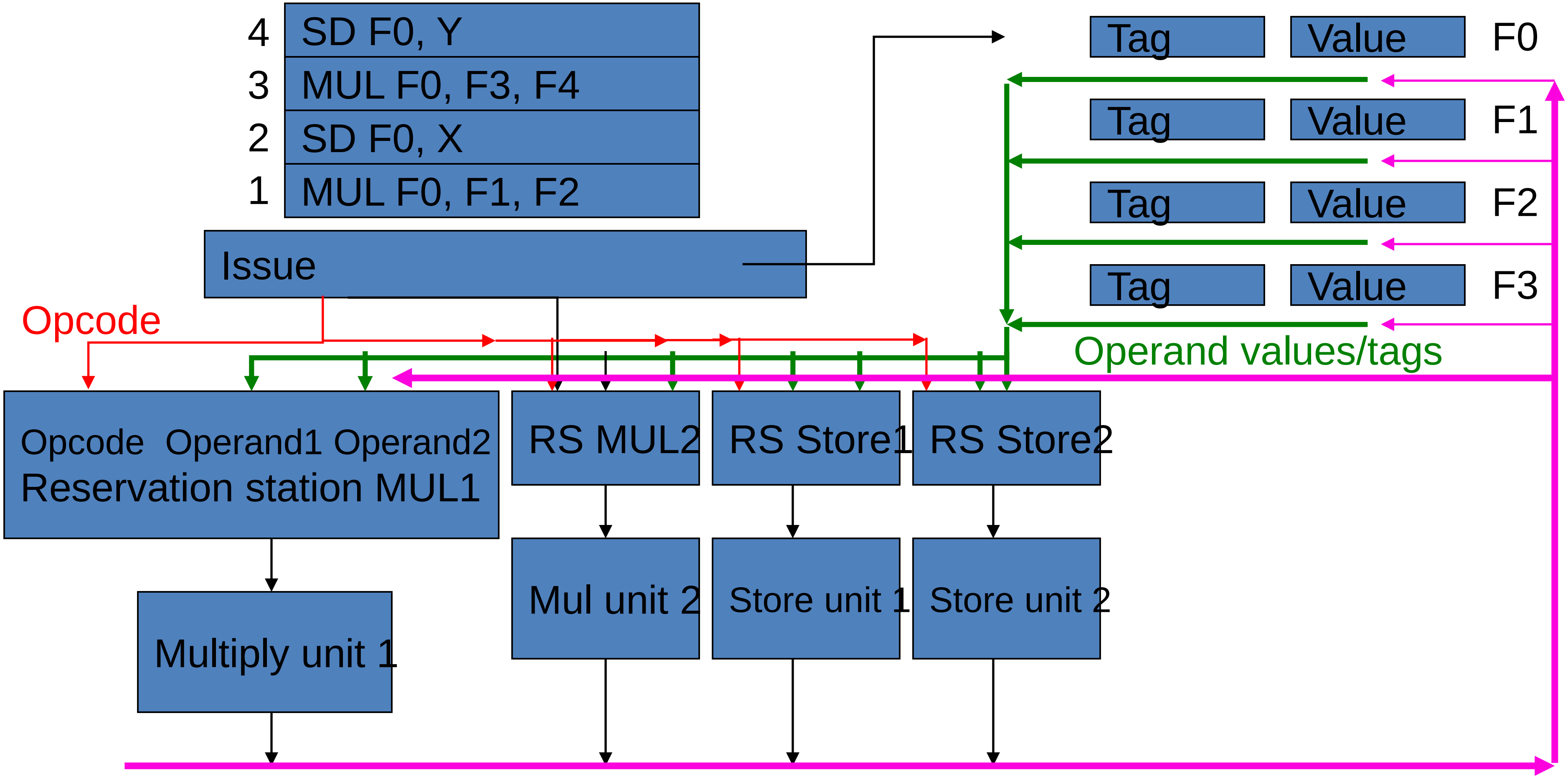


Issue:

- 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

(Copy of earlier slide)

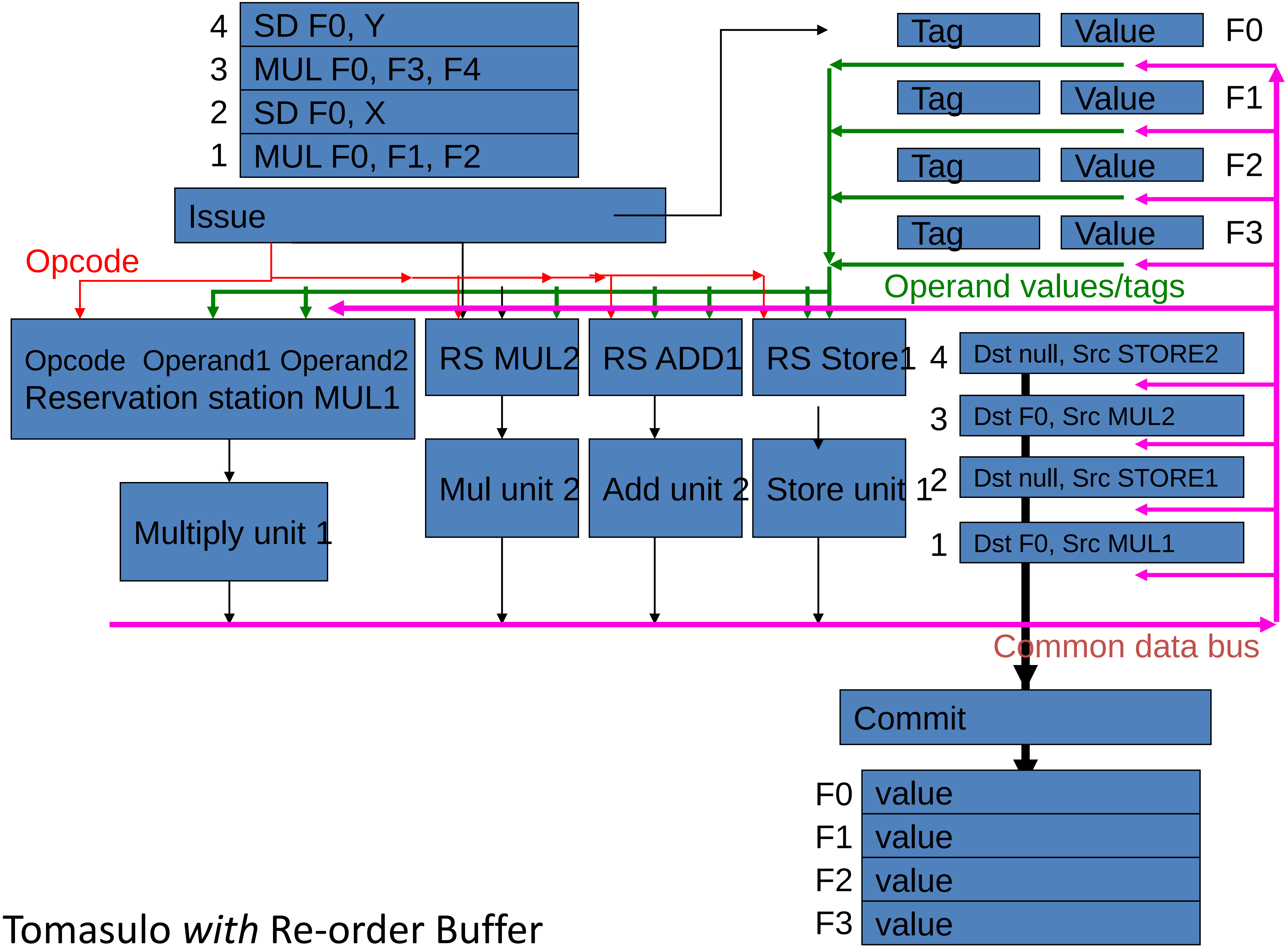
Tomasulo without Re-order Buffer



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

(Copy of earlier slide)



4 SD F0, Y

3 MUL F0, F3, F4

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

4 Dst null, Src STORE2

3 Dst F0, Src MUL2

2 Dst null, Src STORE1

1 Dst F0, Src MUL1

Common data bus

Commit

F0 value

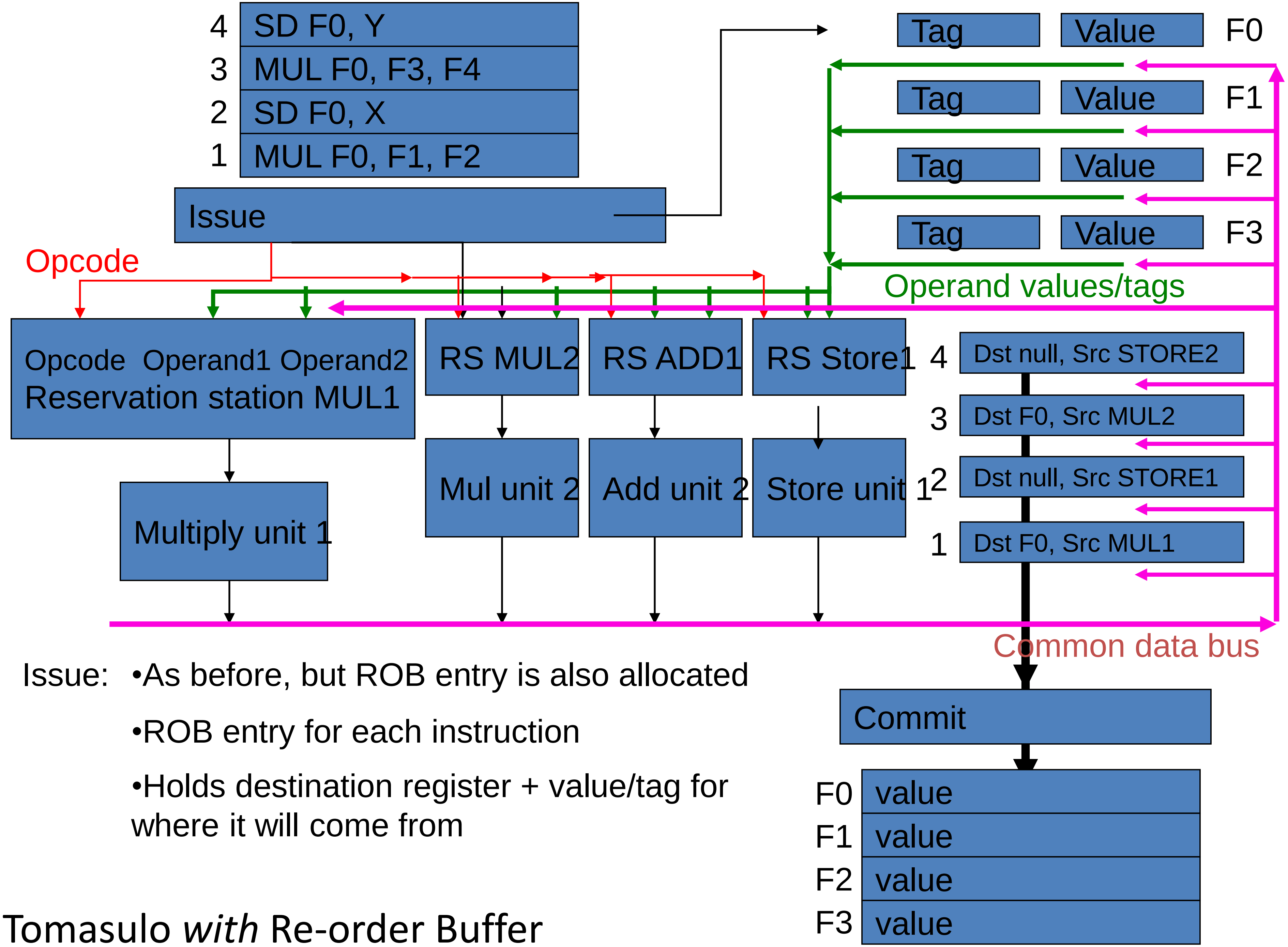
F1 value

F2 value

F3 value

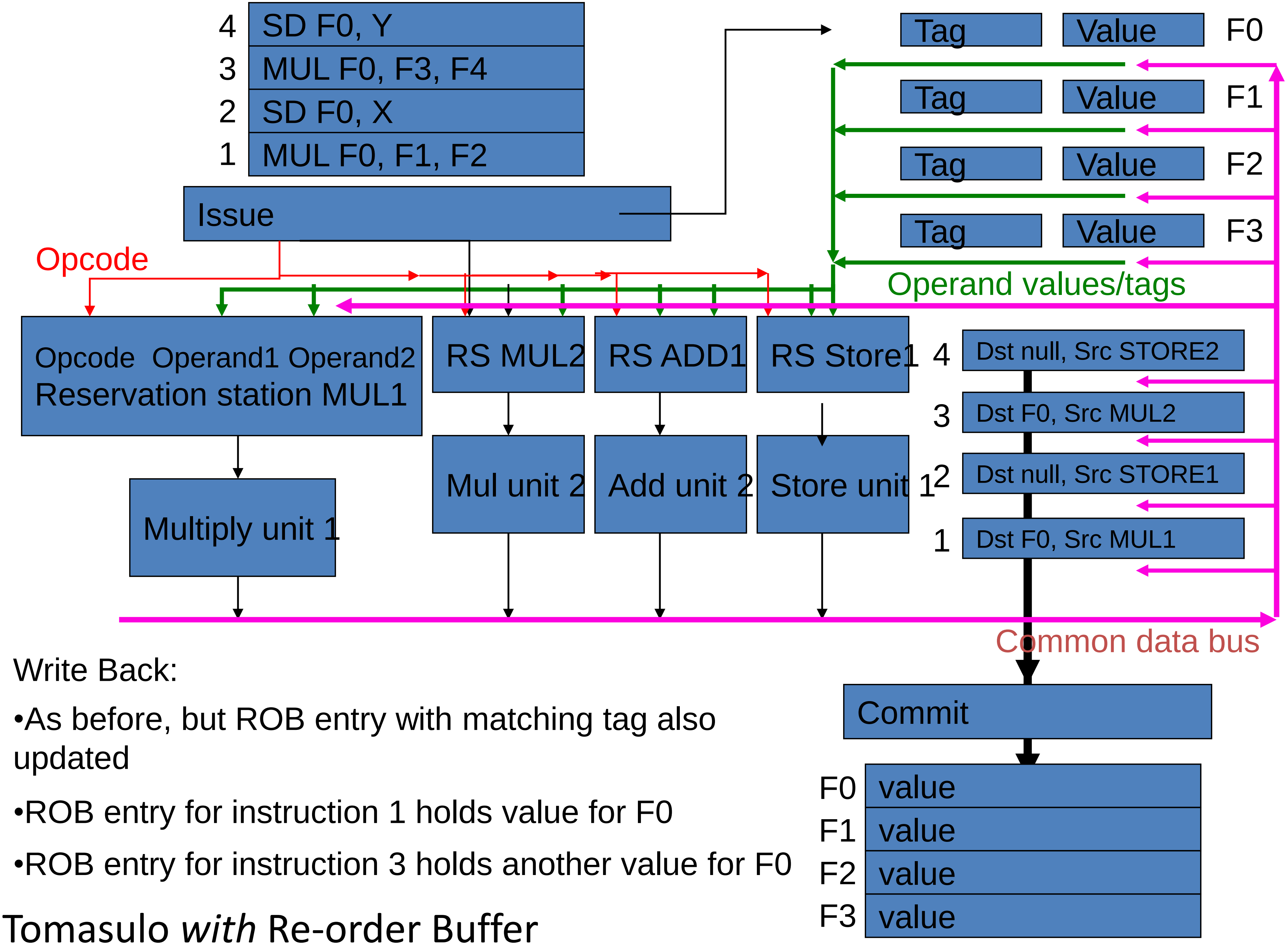
Tomasulo *with* Re-order Buffer

Tomasulo *with* Re-order Buffer



Issue:

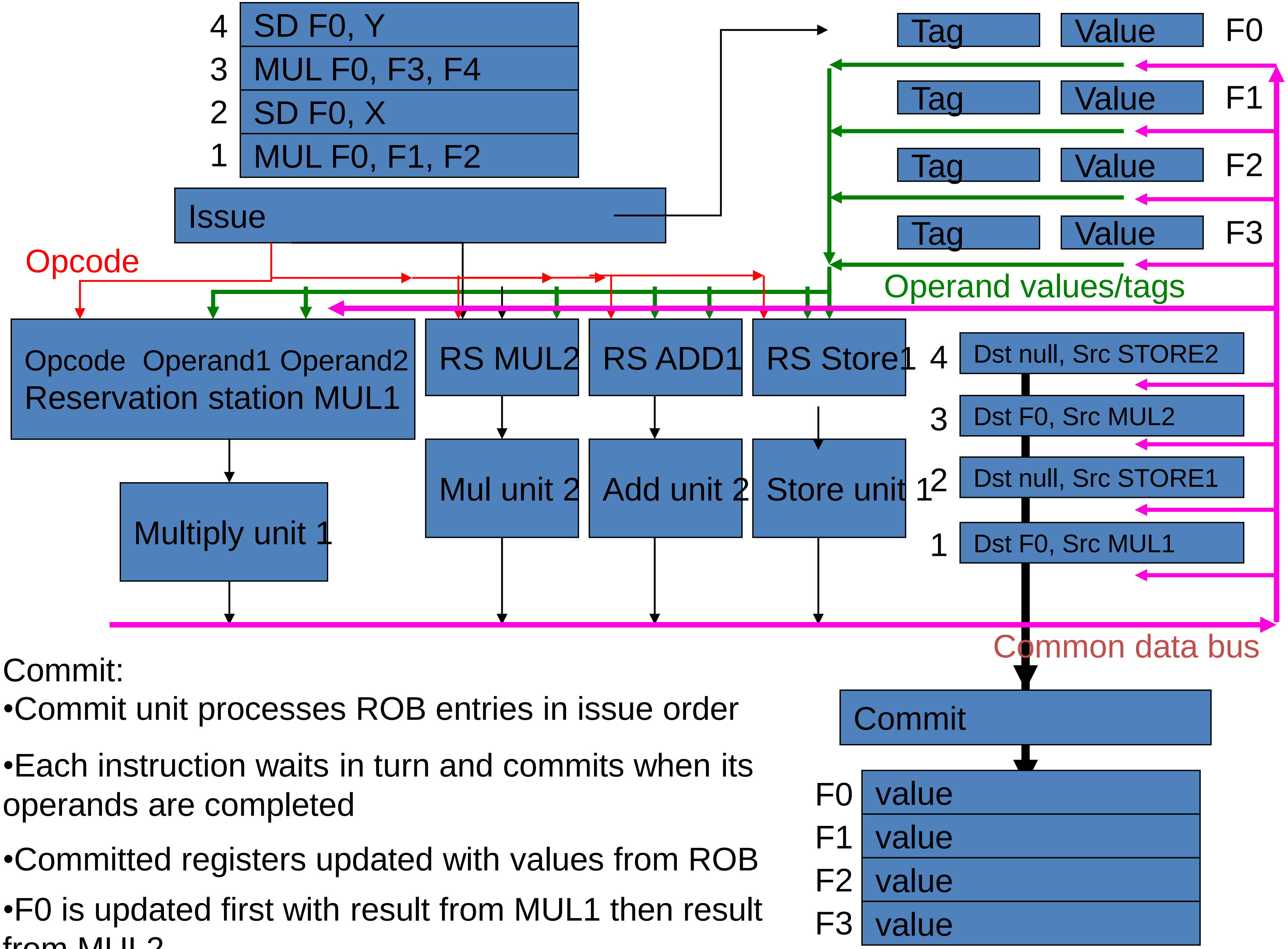
- As before, but ROB entry is also allocated
- ROB entry for each instruction
- Holds destination register + value/tag for where it will come from



Write Back:

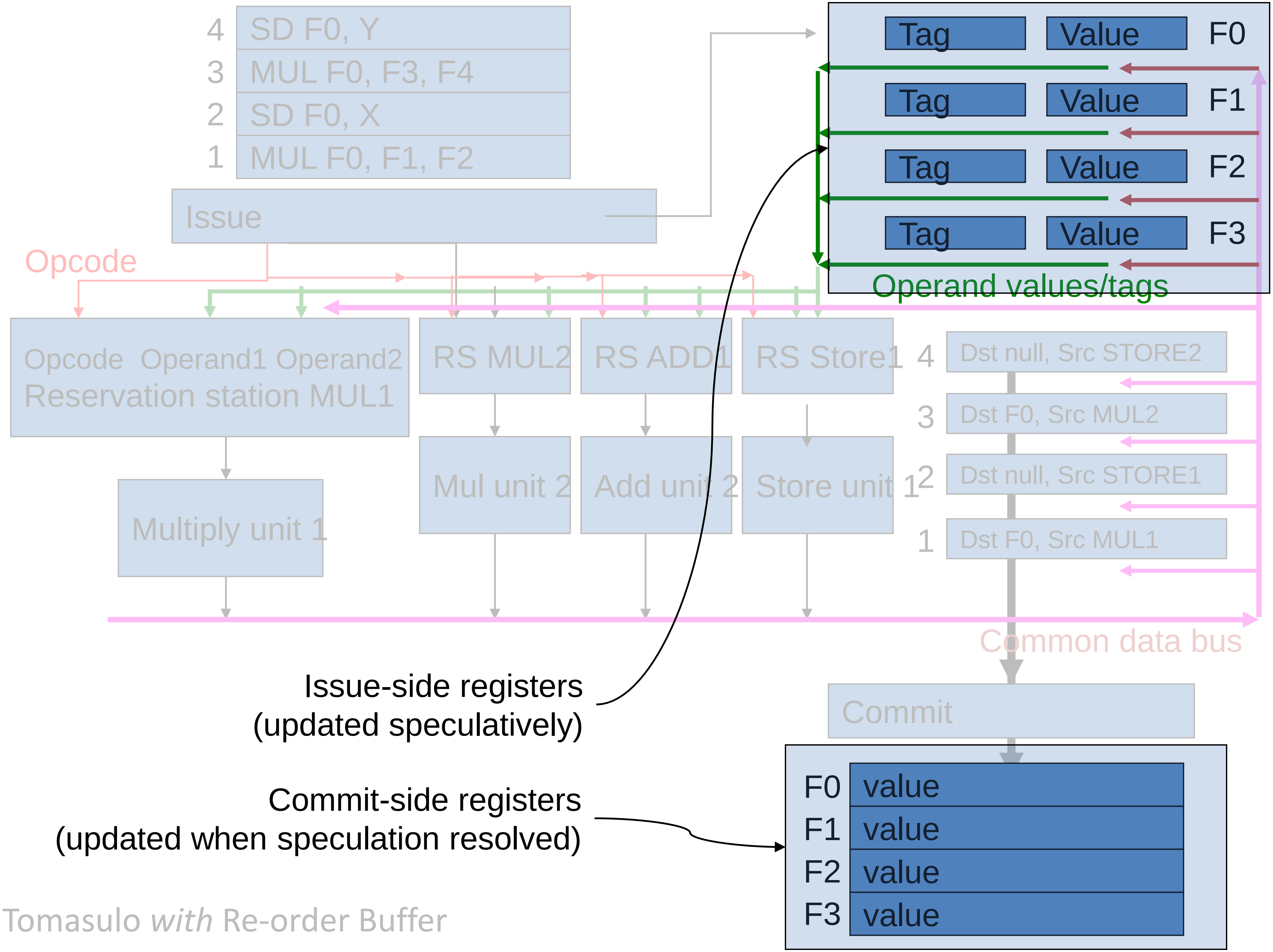
- As before, but ROB entry with matching tag also updated
- ROB entry for instruction 1 holds value for F0
- ROB entry for instruction 3 holds another value for F0

Tomasulo *with* Re-order Buffer



Commit:

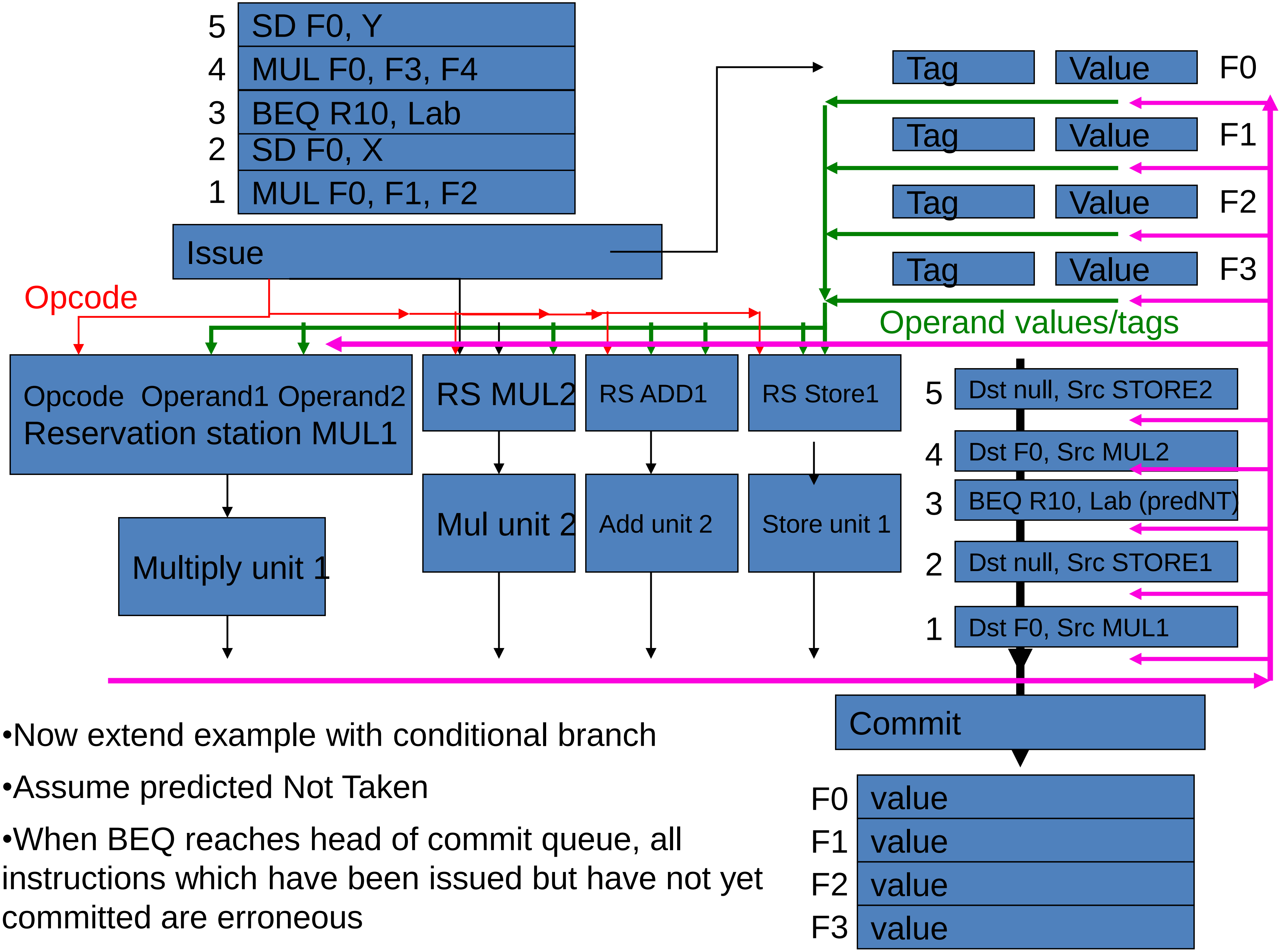
- Commit unit processes ROB entries in issue order
- Each instruction waits in turn and commits when its operands are completed
- Committed registers updated with values from ROB
- F0 is updated first with result from MUL1 then result from MUL2



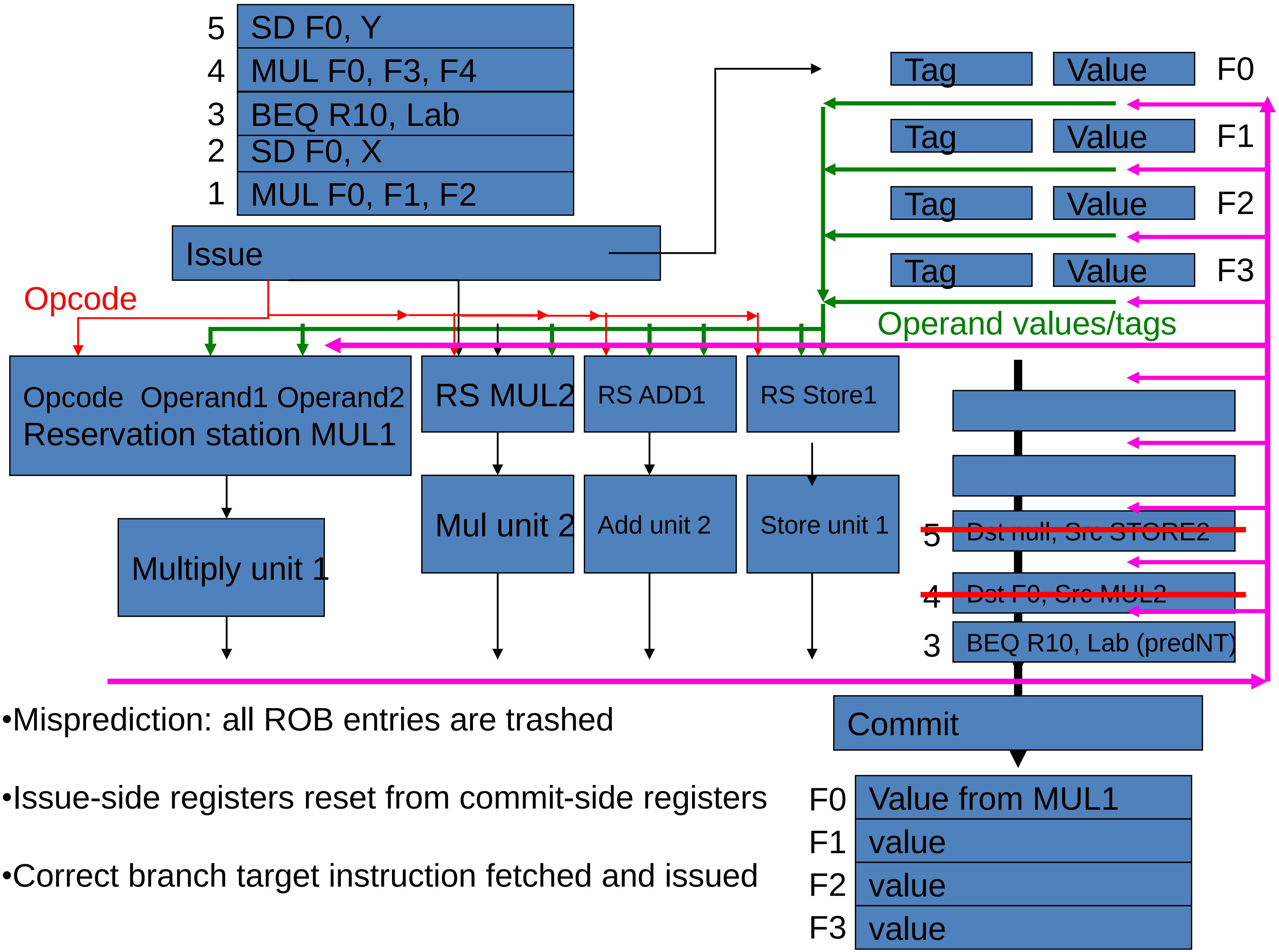
Issue-side registers
(updated speculatively)

Commit-side registers
(updated when speculation resolved)

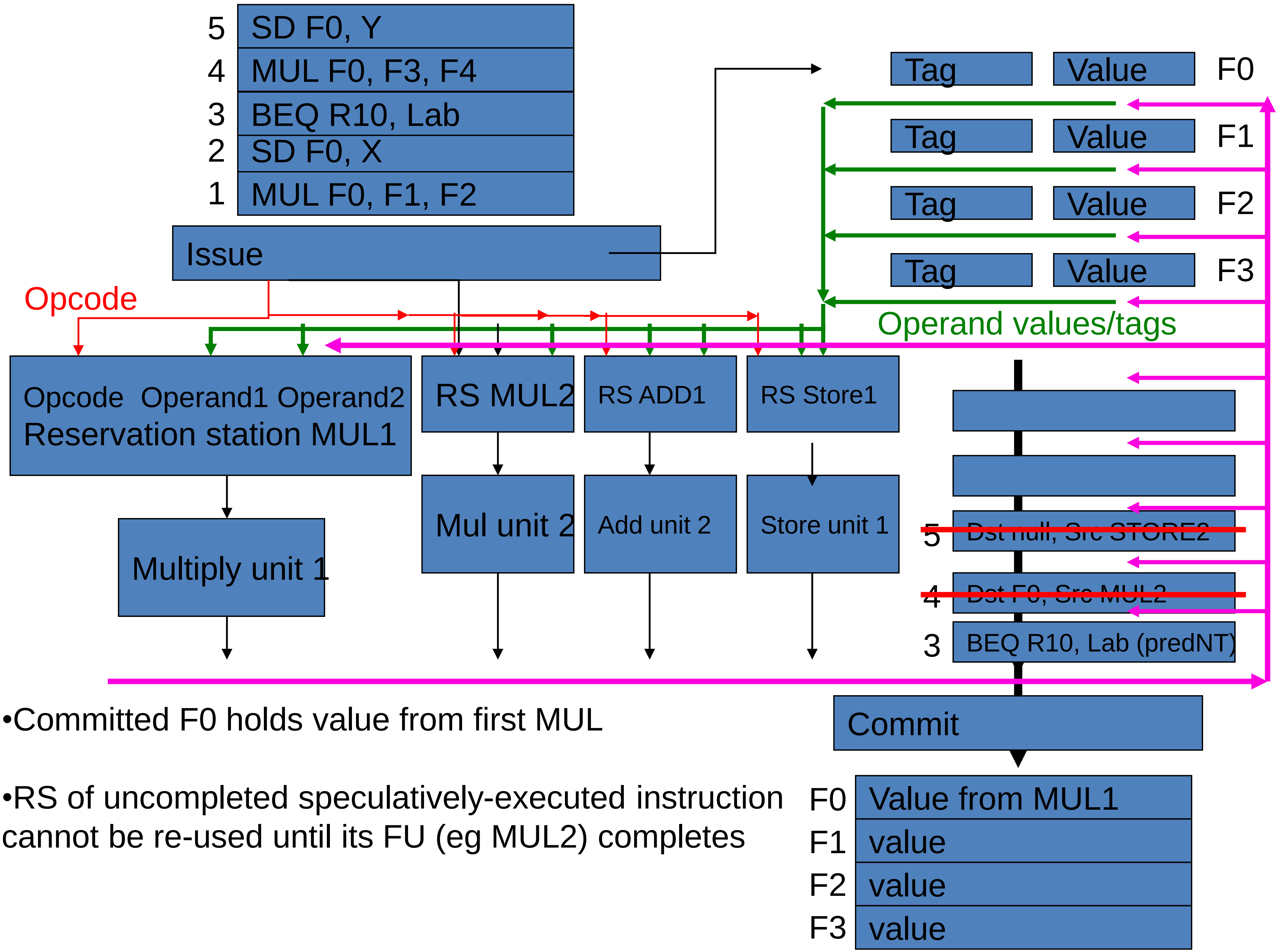
Tomasulo *with* Re-order Buffer



- 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 reset from 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 subleties…

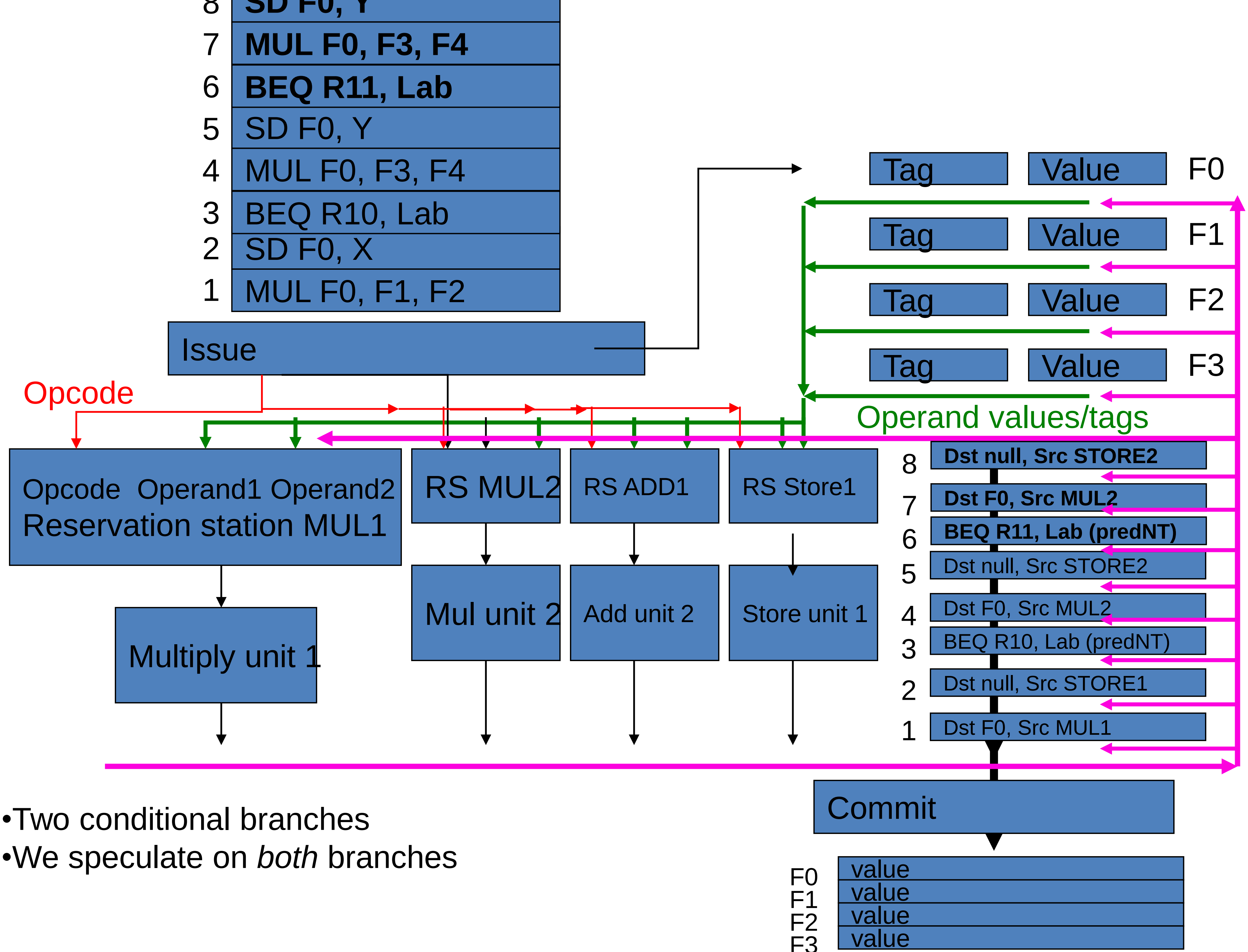
- It’s vital to reduce the branch misprediction penalty. Does the Tomasulo+ROB scheme described here roll-back as soon as the branch is found to be mispredicted?

- 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:

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

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

- 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



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	
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)

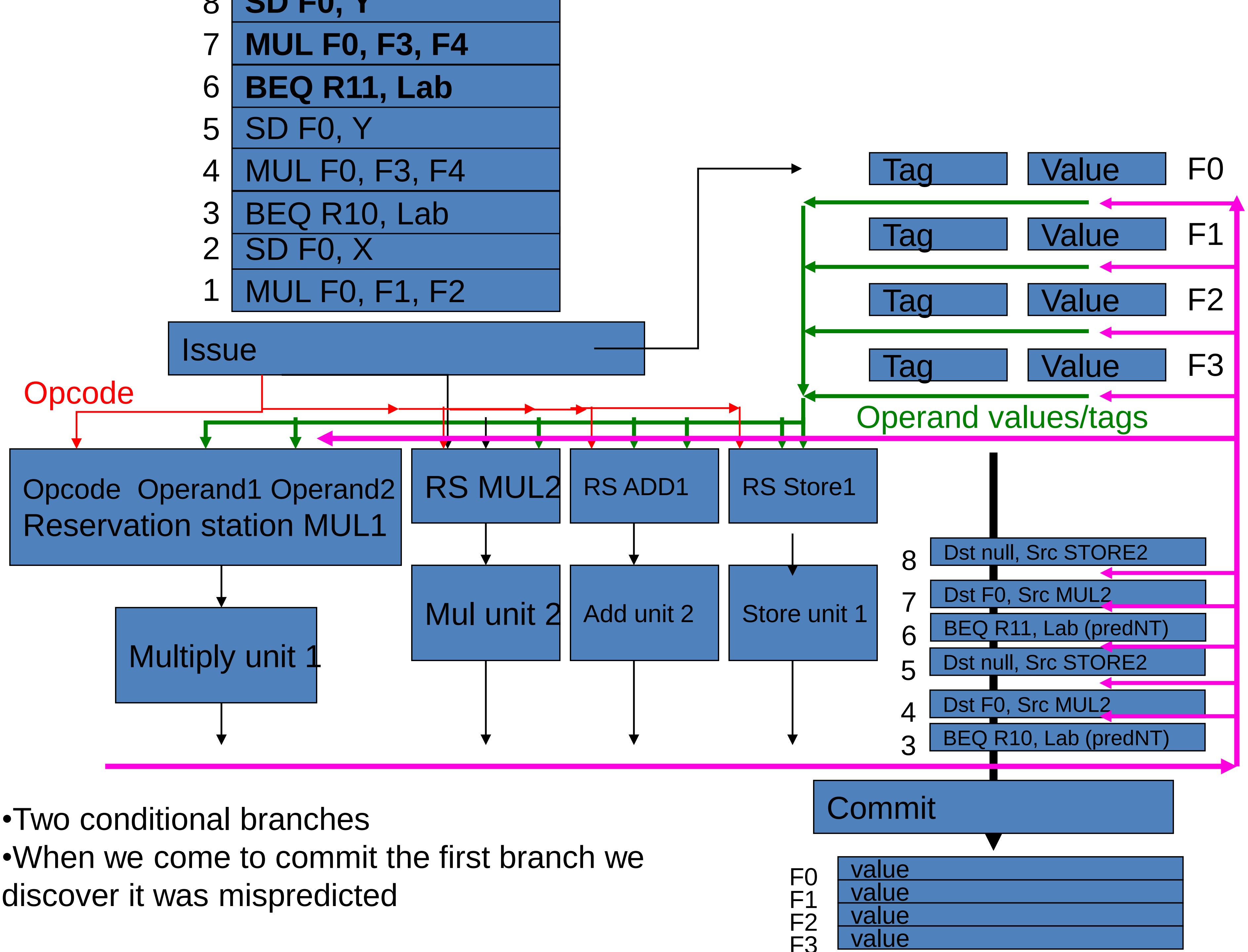
2 Dst null, Src STORE1

1 Dst F0, Src MUL1

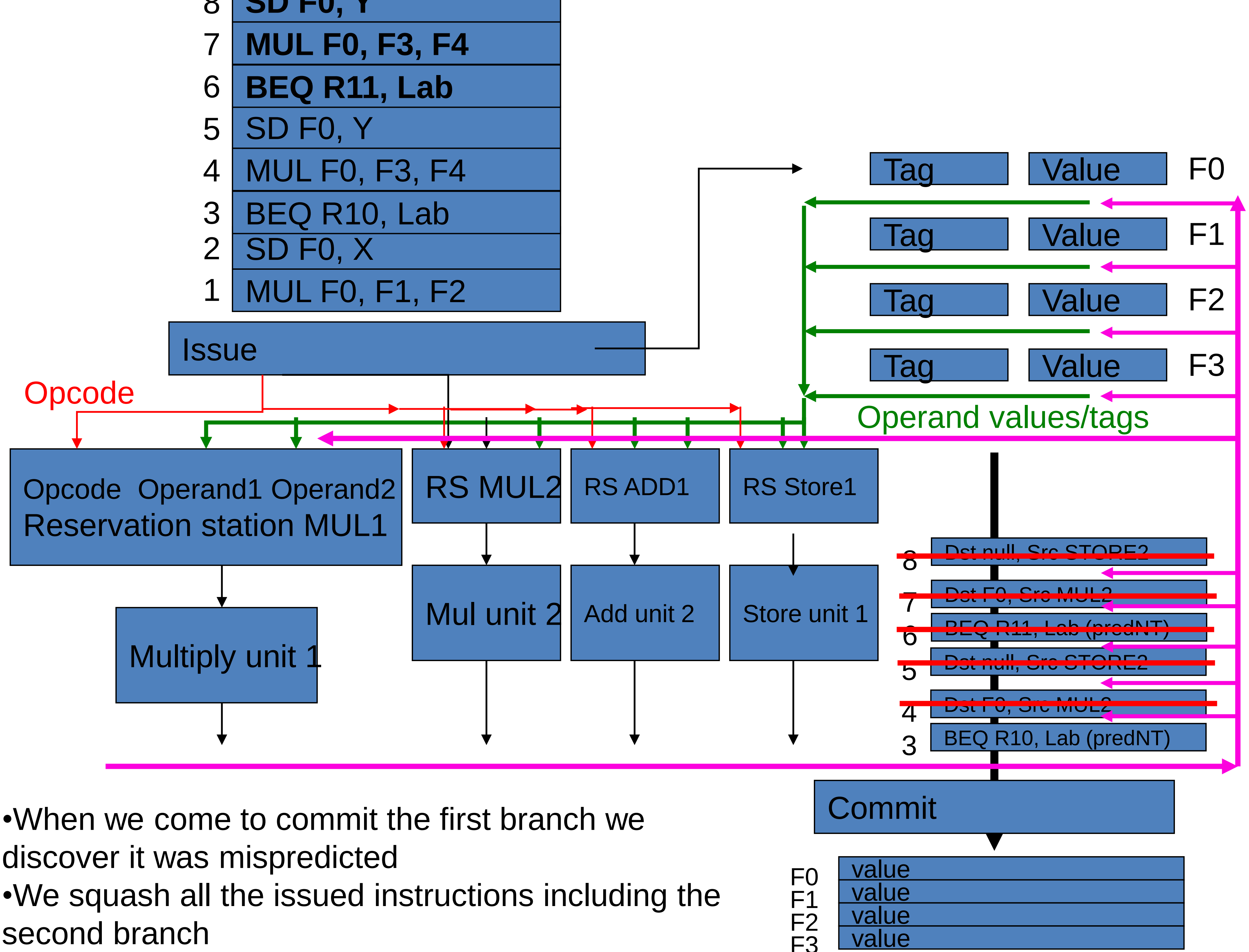
Commit

F0	value
F1	value
F2	value
F3	value

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



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



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

Tomasulo + ROB: Summary

- Reservations stations: *implicit register renaming* to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards of Scoreboard (see textbook)
  - Allows loop unrolling in HW
- Not limited to basic blocks (integer units gets ahead, beyond branches)
- Today, helps cache misses as well
  - Don’t stall for L1 Data cache miss (insufficient ILP for L2 miss?)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation

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)*

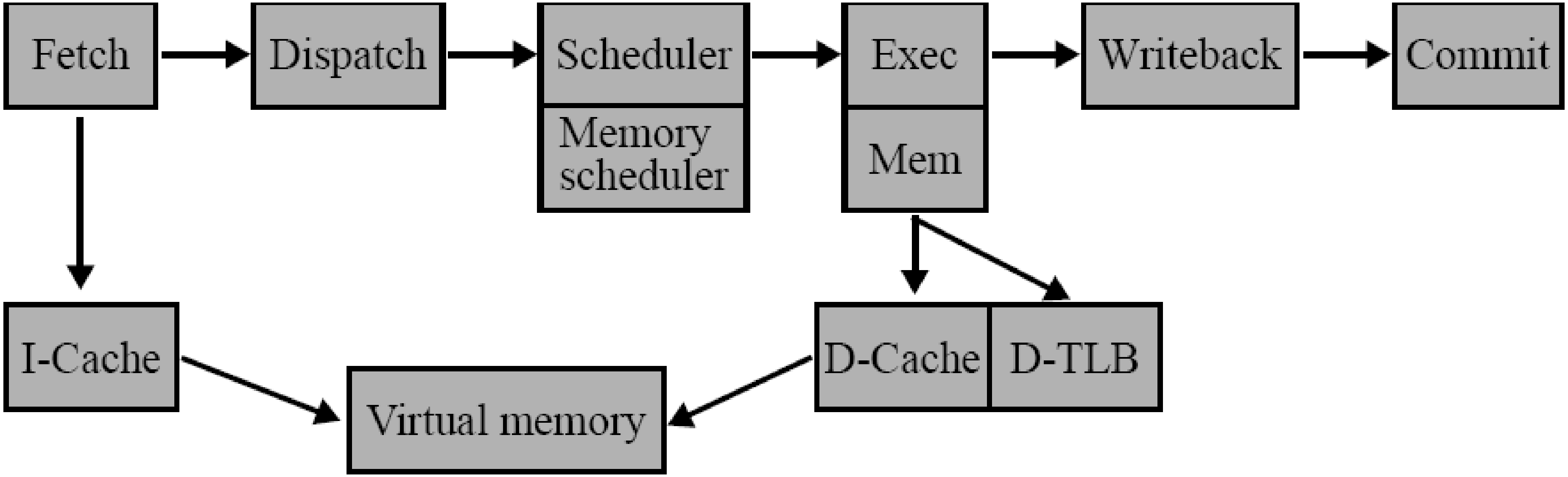
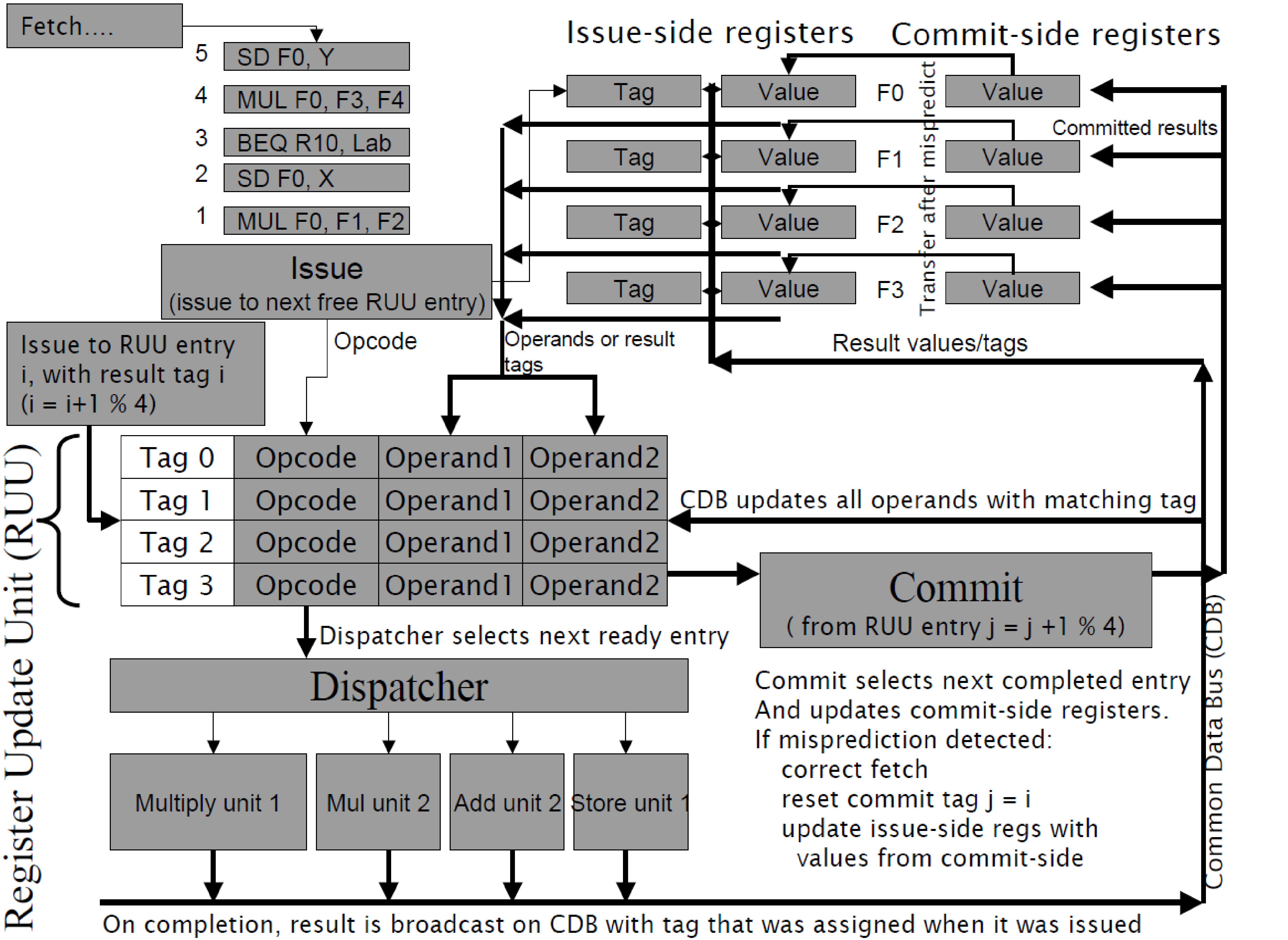


Figure 5. Pipeline for sim-outorder

- 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

(issue to next free RUU entry)

Issue to RUU entry i, with result tag i (i = i+1 % 4)

Opcode

Operands or result tags

Issue-side registers

Commit-side registers

Transfer after misspredict

Committed results

Result values/tags

Register	Issue-side Tag	Issue-side Value	Commit-side Value
F0	Tag	Value	Value
F1	Tag	Value	Value
F2	Tag	Value	Value
F3	Tag	Value	Value

Register Update Unit (RUU)

Tag 0	Opcode	Operand1	Operand2
Tag 1	Opcode	Operand1	Operand2
Tag 2	Opcode	Operand1	Operand2
Tag 3	Opcode	Operand1	Operand2

CDB updates all operands with matching tag

Commit

(from RUU entry j = j +1 % 4)

Dispatcher selects next ready entry

Dispatcher

Multiply unit 1

Mul unit 2

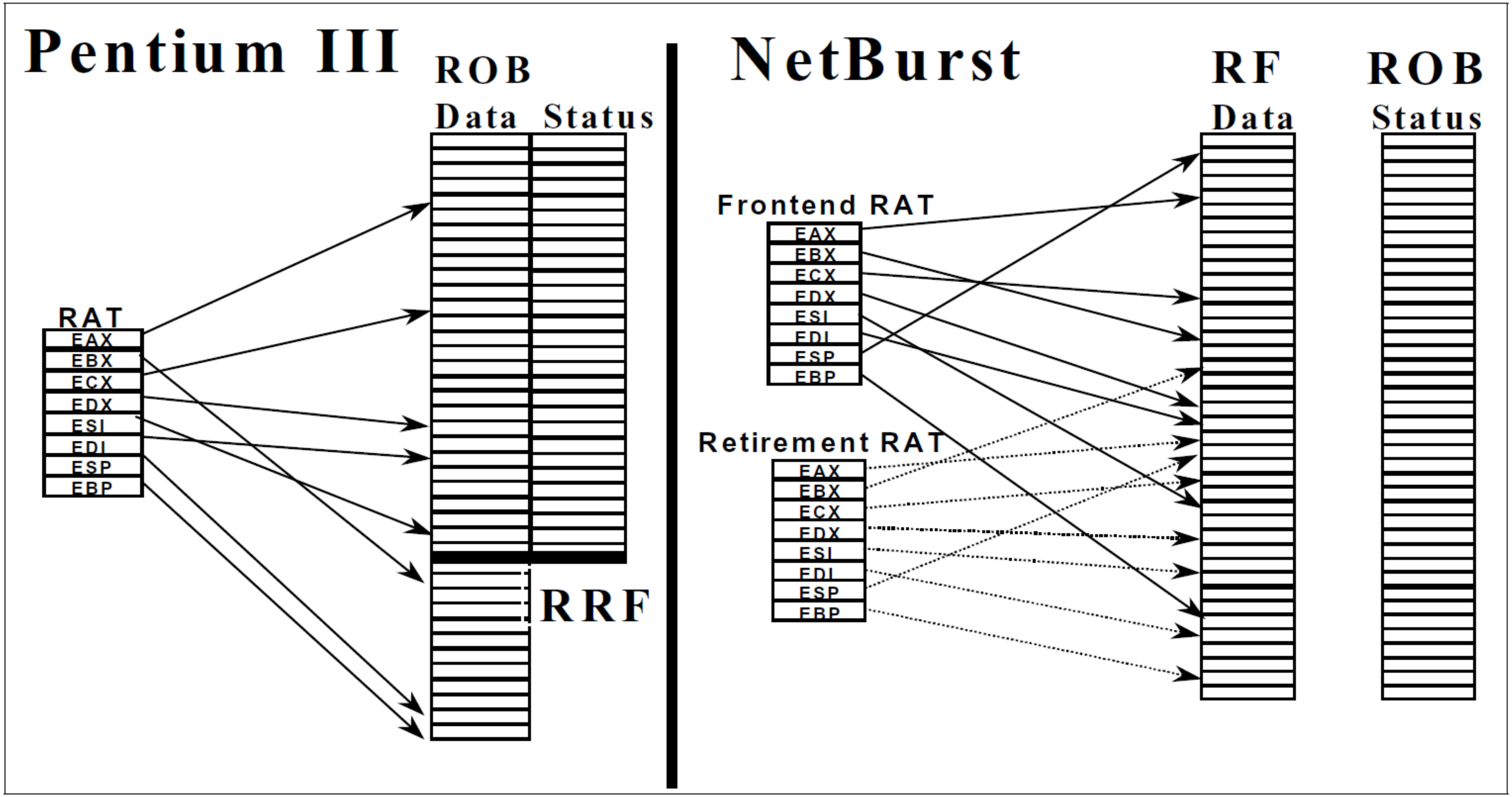
Add unit 2

Store unit 1

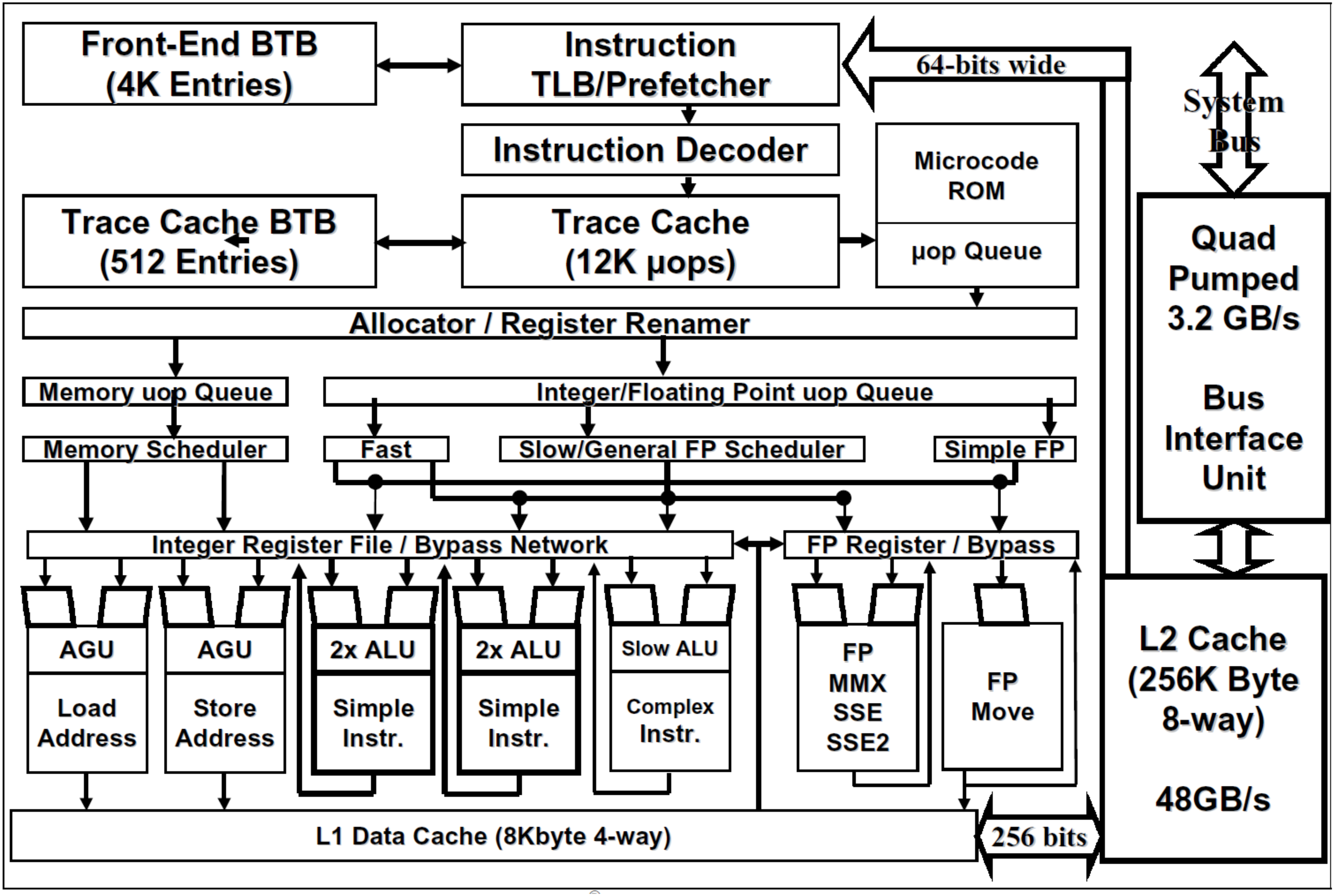
Commit selects next completed entry
And updates commit-side registers.
If misprediction detected:
correct fetch
reset commit tag j = i
update issue-side regs with
values from commit-side

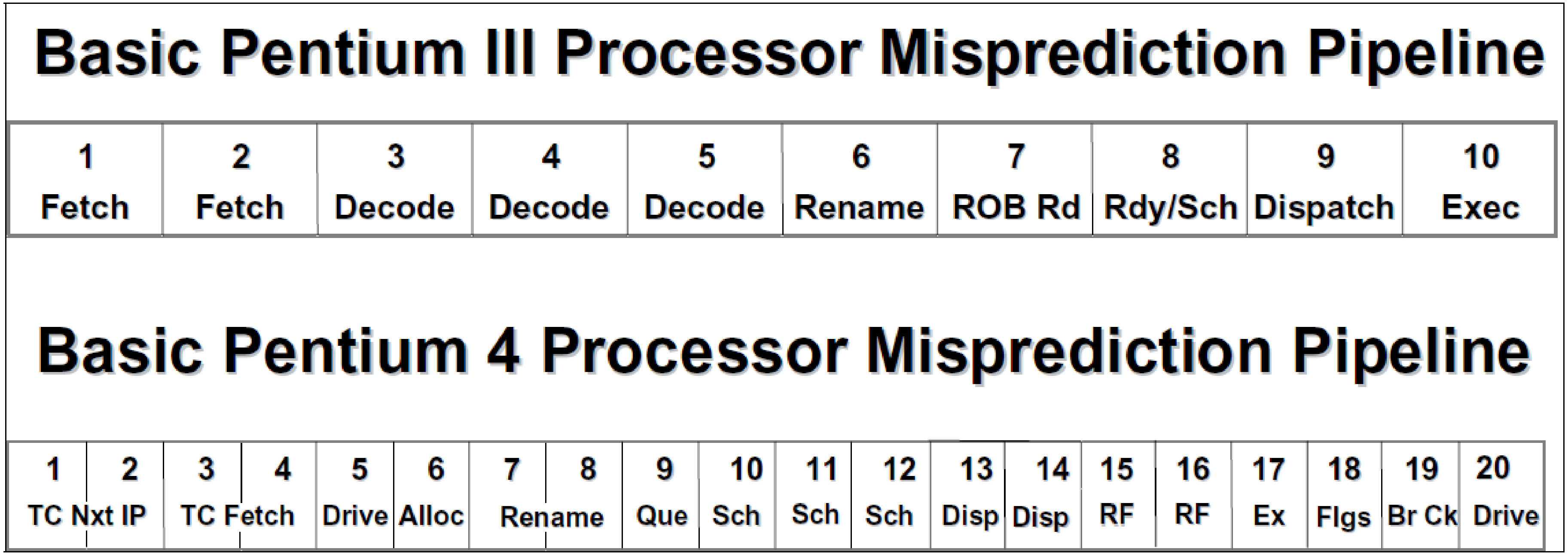
Common Data Bus (CDB)

On completion, result is broadcast on CDB with tag that was assigned when it was issued


Pentium III
ROB
Data
Status
RAT
EAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP
RRF
NetBurst
RF
Data
ROB
Status
Frontend RAT
EAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP
Retirement RAT
EAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP

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