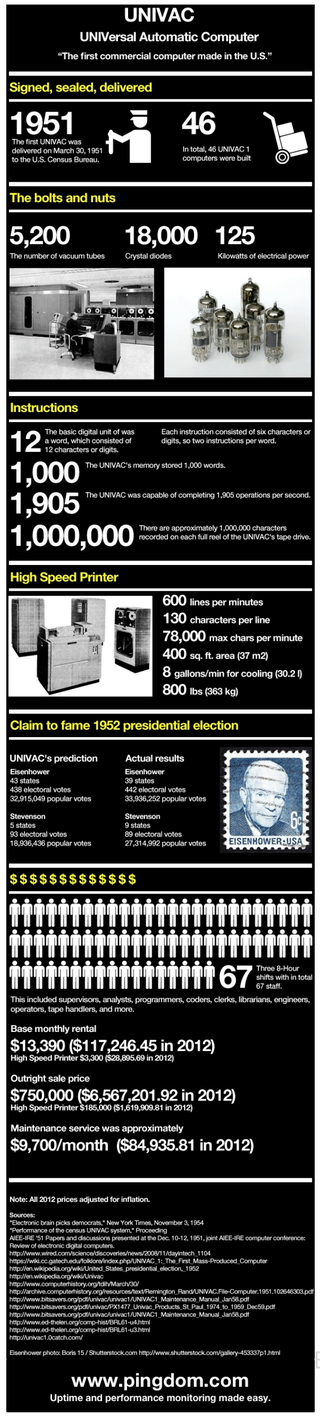
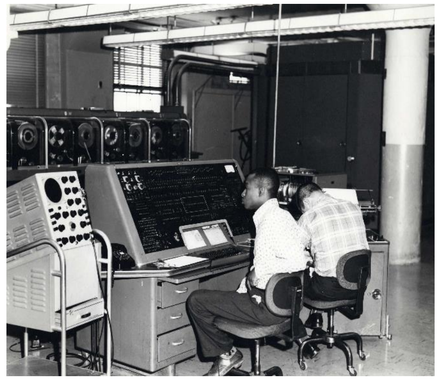
CSE 240A - Spring 2024



1954:
"8 UNIVACs were installed and in operation:

Bureau of the Census, Commerce Dept., Suitland, Maryland
Office of the Air Comptroller, USAF, Washington, D.C.
Army Map Service, U.S. Army, Washington, D.C.
New York University (for Atomic Energy Commission), NY, NY
University of Cal., Radiation Laboratory, Livermore, California
David Taylor Model Basin, U.S.N. Bureau of Ships, Maryland
Prudential Insurance Company
General Electric Company"



https://www.census.gov/history/img/univacphoto.jpg

Ca. 1960

https://www.thocp.net/hardware/univac.htm

Recap: How many cycles per loop iteration now . . .

```
loop: ld     f0, 0(r1)
      muld   f4, f0, f2
      sd     f4, 0(r1)
      addi   r1, r1, 8
      slt    r3, r1, r2
      bnez   r3, loop
      nop
```

was seq

was sgt

```
loop: ld     f0, 0(r1)      1
      addi   r1, r1, 8      2
      muld   f4, f0, f2     3
      slt    r3, r1, r2     4
      bnez   r3, loop       5
      sd     f4, -8(r1)     9 = 3 + 6

      ld     r0, 0(r1)      10
```

Operations	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

A. 6

B. 8

C. 9

D. 10

E. 14

webclicker.web.app : nutcom

Loop Unrolling (1)

```
loop: ld    f0, 0(r1)
      addi  r1, r1, 8
      muld  f4, f0, f2
      slt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
```

```
loop: ld    f0, 0(r1)
      addi  r1, r1, 8
      muld  f4, f0, f2
      slt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
```

Operations	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

Loop Unrolling (2)

```
loop: ld    f0, 0(r1)
      addi  r1, r1, 8
      muld  f4, f0, f2
      slt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
```

```
loop: ld    f0, 0(r1)
      addi  r1, r1, 8
      muld  f4, f0, f2
      slt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
      ld    f0, 0(r1)
      addi  r1, r1, 8
      muld  f4, f0, f2
      slt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
```

Operations	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

Loop Unrolling (3)

```
loop: ld    f0, 0(r1)
      addi  r1, r1, 8
      muld  f4, f0, f2
      slt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
```

Eliminate redundant code

```
loop: ld    f0, 0(r1)
      addi  r1, r1, 8
      muld  f4, f0, f2
      slt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
      ld    f0, 0(r1)
      addi  r1, r1, 8
      muld  f4, f0, f2
      slt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
```

Operations	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

Loop Unrolling (4)

```
loop: ld    f0, 0(r1)
      addi  r1, r1, 8
      muld  f4, f0, f2
      sgt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
```

```
loop: ld    f0, 0(r1)
      muld  f4, f0, f2
      sd    f4, 0(r1)
      ld    f0, 8(r1)
      addi  r1, r1, 16
      muld  f4, f0, f2
      sgt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
```

Operations	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

Register Renaming(1)

```
loop: ld     f0, 0(r1)
      addi   r1, r1, 8
      muld   f4, f0, f2
      sgt    r3, r1, r2
      bnez   r3, loop
      sd     f4, -8(r1)
```

```
loop: ld     f0, 0(r1)
      muld   f4, f0, f2
      sd     f4, 0(r1)
      ld     f0, 8(r1)
      addi   r1, r1, 16
      muld   f4, f0, f2
      sgt    r3, r1, r2
      bnez   r3, loop
      sd     f4, -8(r1)
```

Operations	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

- Higher latency operations, LD and MULD
- Use more registers so we can reschedule code.

Register Renaming(2)

```
loop: ld    f0, 0(r1)
      addi  r1, r1, 8
      muld  f4, f0, f2
      sgt   r3, r1, r2
      bnez  r3, loop
      sd    f4, -8(r1)
```

```
loop: ld    f0, 0(r1)
      muld  f4, f0, f2
      sd    f4, 0(r1)
      ld    f10, 8(r1)
      addi  r1, r1, 16
      muld  f14, f10, f2
      sgt   r3, r1, r2
      bnez  r3, loop
      sd    f14, -8(r1)
```

Operations	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

- After renaming f0->f10, f4->f14 can reschedule code

Register Renaming/ Reschedule (3)

```
loop: ld    f0, 0(r1)
      muld  f4, f0, f2
      sd    f4, 0(r1)
      ld    f10, 8(r1)
      addi  r1, r1, 16
      muld  f14, f10, f2
      sgt   r3, r1, r2
      bnez  r3, loop
      sd    f14, -8(r1)
```

```
loop: ld   f0, 0(r1)
      ld   f10, 8(r1)
      muld f4, f0, f2
      muld f14, f10, f2
      addi r1, r1, 16
      sgt  r3, r1, r2
      sd   f4, 0(r1)
bnez  r3,  loop
      sd   f14, -8(r1)
```

Operations	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

Register Renaming/ Reschedule (Final)

```
loop: ld   f0, 0(r1)          1
      ld   f10, 8(r1)         2
      muld f4, f0, f2         3
      muld f14, f10, f2       4
      addi r1, r1, 16         5
      sgt  r3, r1, r2         6
      sd   f4, 0(r1)          9 = 3 + 6
bnez  r3,  loop              10
      sd   f14, -8(r1)       11 = max(11, 4 + 6)

      ld   f0, 0(r1)         12
```

Operations	Latency (stalls)
FP Mult	6 (5)
LD	2 (1)
Int ALU	1 (0)

11 cycles for 2 iterations
5.5 cycles per iteraction

Limitations on Loop Unrolling

1.

2.

3.

Compiler Perspectives on Code Movement

- *dependencies* are a property of the code. Whether they generate a hazard is a property of the HW implementation.
- *Compiler* must respect TRUE data dependencies (RAW)
  - *registers* - fairly easy to identify
  - *Memory* – hard for the compiler
    - Does `100(r4) == 20(r6)` ?
    - Between loop iterations does `4(r4) == 4(r4)` ?
- False dependencies (WAR and WAW) can sometimes be fixed (register renaming)

Compilers and Control Dependence

Compilers must preserve *control dependence*

```
if (c1)
   I1;
if (c2)
   I2;
```

I1 depends on c1,

I2 depends on c2, but not c1

Restricting on code motion

- An instruction that ***control dependent*** *on a branch* cannot generally be moved before the branch
- An instruction that **not *control dependent*** on a branch cannot generally be moved after the branch.
- Control dependencies can be relaxed to get parallelism; as long as we get the same effect
  - we preserve exception and data flow.

```
I0;
if (c1)
   I1;
else
   I2;
```

→

```
if (c1) {
   I0;
   I1;
} else {
   I0;
   I2;
}
```

Code Motion

- Can be done in SW or HW
  - Why SW?
  - next : Why HW?

- Want hardware to be able to
  - move instructions across branches
  - ignore or overcome false depedencies

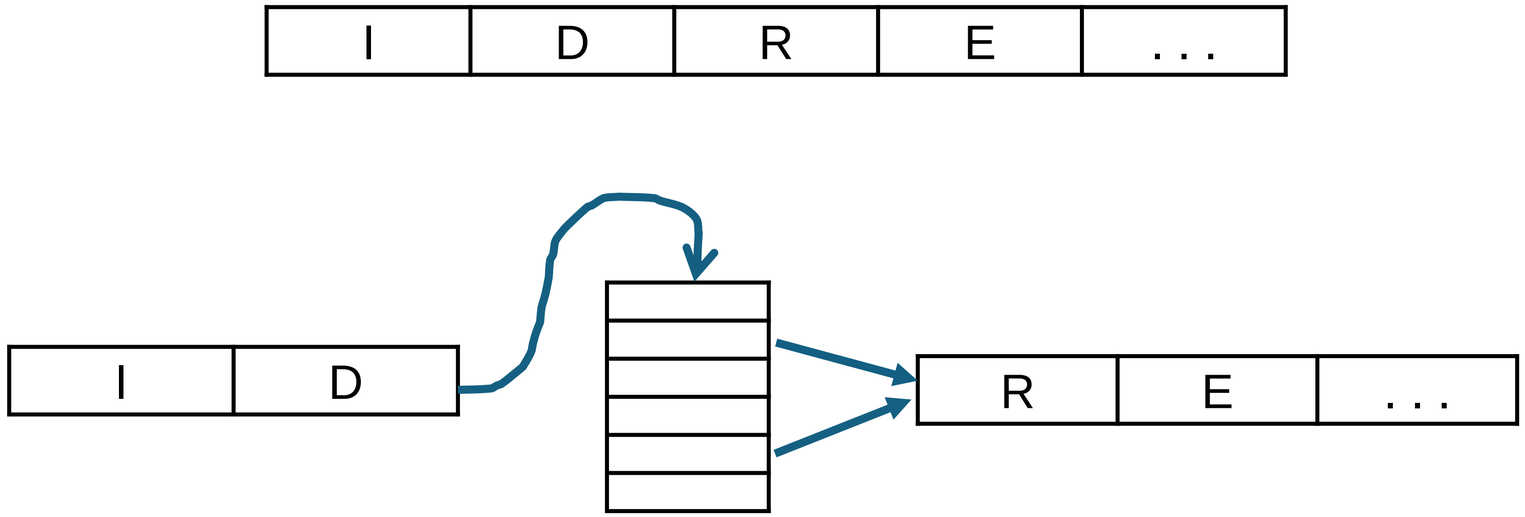
Benefits of HW schemes

- HW uses runtime information
  - works when we can't know dependence until runtime
    - variable latency (e.g. cache hit, cache miss)
    - control dependence (what is the branch doing or likely to do)
    - memory dependencies (values passed thru memory)
  - can schedule different every time thru the code based on runtime information
  - Compiler changes (see upcoming slide)
  - *Code doesn't need to be optimized for a particular microarchitecture*
- *Key Ideas Allow instructions behind a stall to proceed.*
  - out of order execution -> out of order completion (?)

```
divd f0, f2, f4
addd f10, f0, f8
subd f12, f8, f14
```

The Problem with In-Order

- Instructions get stuck in the I D stage
  - dependency
  - structural hazard
- What if we can put the instruction waiting for a dependency or hazard resolution off to the side



We split D and R
Some machines may not

Dynamic Scheduling by hand

In – order

Out – of order

```
div.d f0, f2, f4   // 10 cycles
add.d f10, f0, f8  // 4 cycles
sub.d f12, f8, f14 // 4 cycles
add.d f20, f2, f3
mul.d f13, f12, f2 // 6 cycles
add.d f4, f1, f3
add.d f5, f4, f13
```

assume f FP ADD/SUB is fully pipelines

Which statements are likely true of out-of-order execution

A. makes the compilation harder and hardware exception handling harder

B. makes the compilation harder and hardware exception handling easier

C. makes the compilation easier and hardware exception handling harder.

D. makes the compilation easier and hardware exception handling easier.

webclicker.web.app : nutcom

What's the deal with exceptions

- In order
  - Exception PC indicates where all prior instructions have completed and all subsequent instructions have not.
- Out-of-Order
  - if instructions complete out of order, some instructions may have completed but should not have based in where the exception is taken (exception PC)
  - recently – side effects of Out-of-Order have created security holes (speculation … later)

Key Points

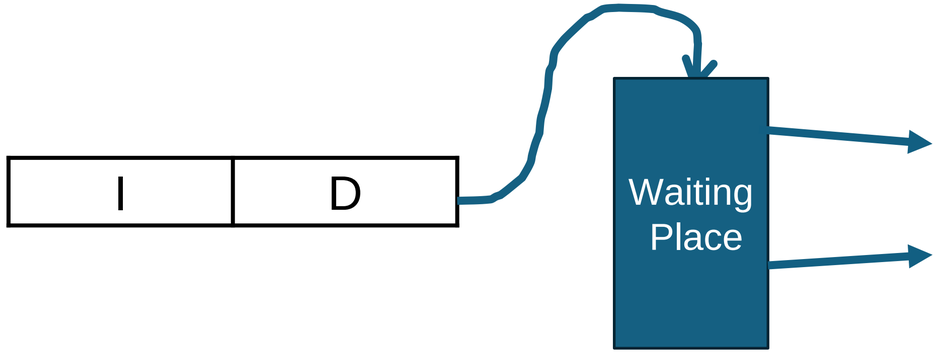
- Both Hardware and Software can find, create and exploit ILP
- Loop level parallelism is usually easiest to see
- Dependencies exist in programs -> become hazards if HW cannot resolve
- SW dependencies/compiler sophistication determine if compiler can/should unroll loops
- SW code motion is limited by lack of runtime dependencies (especially memory), latencies (especially memory) and control flow.

Hardware Dynamic Scheduling

The Problem

- How do we allow `add` to go around `fadd`?
- if ID stage remains in order, then instructions must leave ID stage even if their operands are not ready
- "Waiting Place"
  - place to keep instructions which are waiting for operands to be ready
  - need someway to signal waiting instructions when their operands become ready
- Instructions Issue to execution when they are ready
- Need someway to put instructions back in order
  - exceptions
  - branch recovery

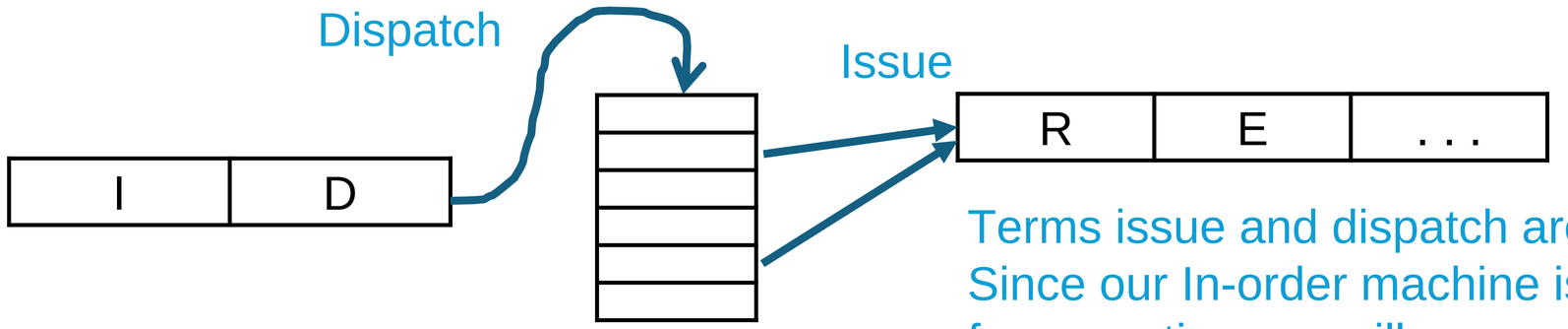
```
fdiv f2, f4, f6
fadd f8, f2, f4
add  r9, r2, r1
sub  r6, f3, f1
```



First HW ILP Technique : Out-of-order Issue/Dynamic Scheduling

- Issue – we will use the definition of sending an instruction to a functional unit for execution
- Separate detection of structural hazards from data hazards
- Split ID into two:
  - Dispatch (decode, check for structural hazards) [e.g. "waiting place" is full]
  - Read operands when no data hazards
- Must be able to dispatch even when data hazards exist.

Instructions dispatch in order but issue (proceed to execute) potentially out of order



Terms issue and dispatch are not universal
Since our In-order machine issued instructions for execution, we will continue to use that term

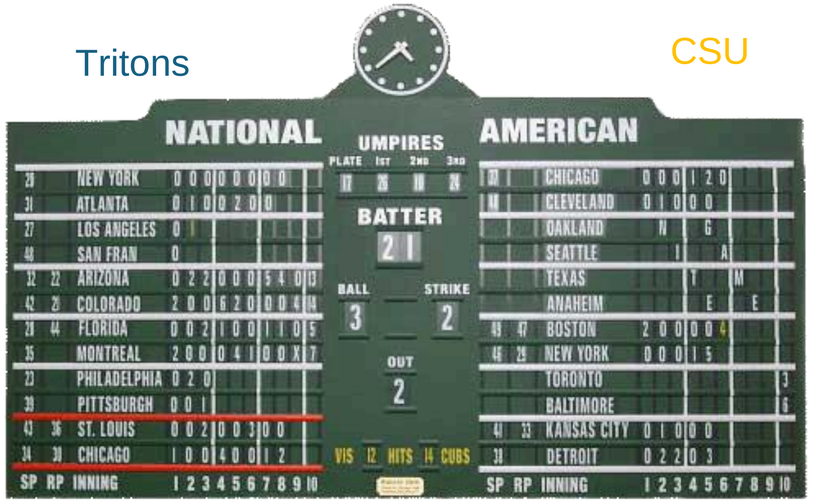
Scoreboard (courtesy of cse260)

Table that keeps a status for each operand

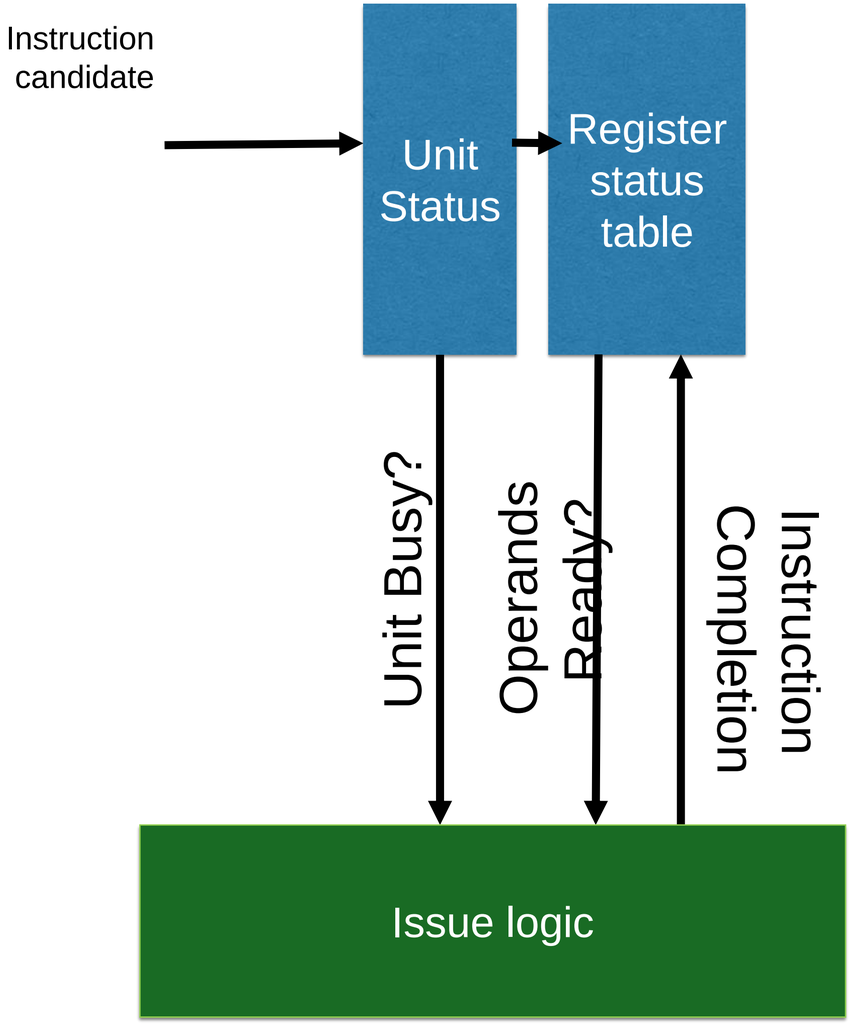
Instructions can check scoreboard to see if operand has a pending update or is valid.

Instructions don't issue unless operands are available.

Scoreboard is updated when FU completes and instruction (writes register file)



http://www.scoreboardclassics.com/images/wrigley-field-scoreboard.gif



Dynamic Scheduling (out-of-order execution) - some history

- CDC 6600 scoreboard
  - instruction storage added to each function execution unit
  - instructions Dispatch to the FU when no structural hazards
  - instructions Issue when dependencies are satisfied.
  - Instructions in different FU's can execute out of order.
  - scoreboard tracks RAW, WAR, WAW hazards and informs when an instruction can proceed.
  - No forwarding in the FU pipelines (pipes are short)
  - No register renaming – wait for WAR and WAW hazards
- Tomasulo (IBM 360/91) [reservation stations]
- Instruction Queue (MIPS R10K, Alpha 21264, …)

CDC 6600

Tomasulo Algorithm

- IBM 360/91 (c 1964) – about 3 years after CDC6600
- Goal High Performance w/o special compilers
- Difference from CDC
  - IBM had only 2 register specifiers (CDC had 3)
  - IBM had 4 FP registeres (CDC had 8)
  - Why does that matter?

IBM 360/91

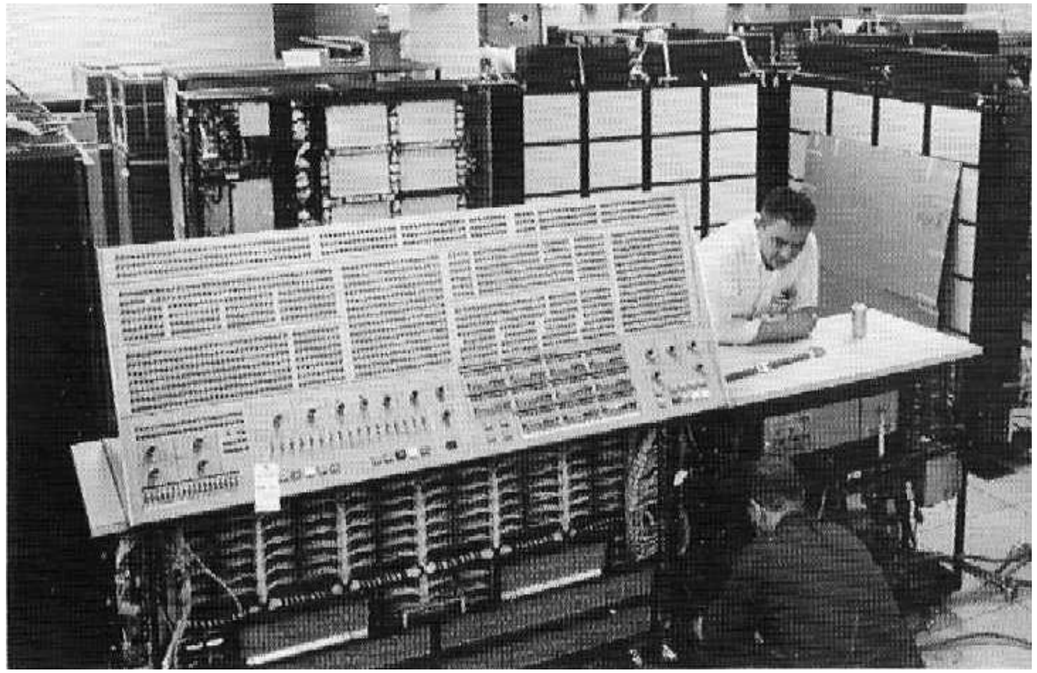


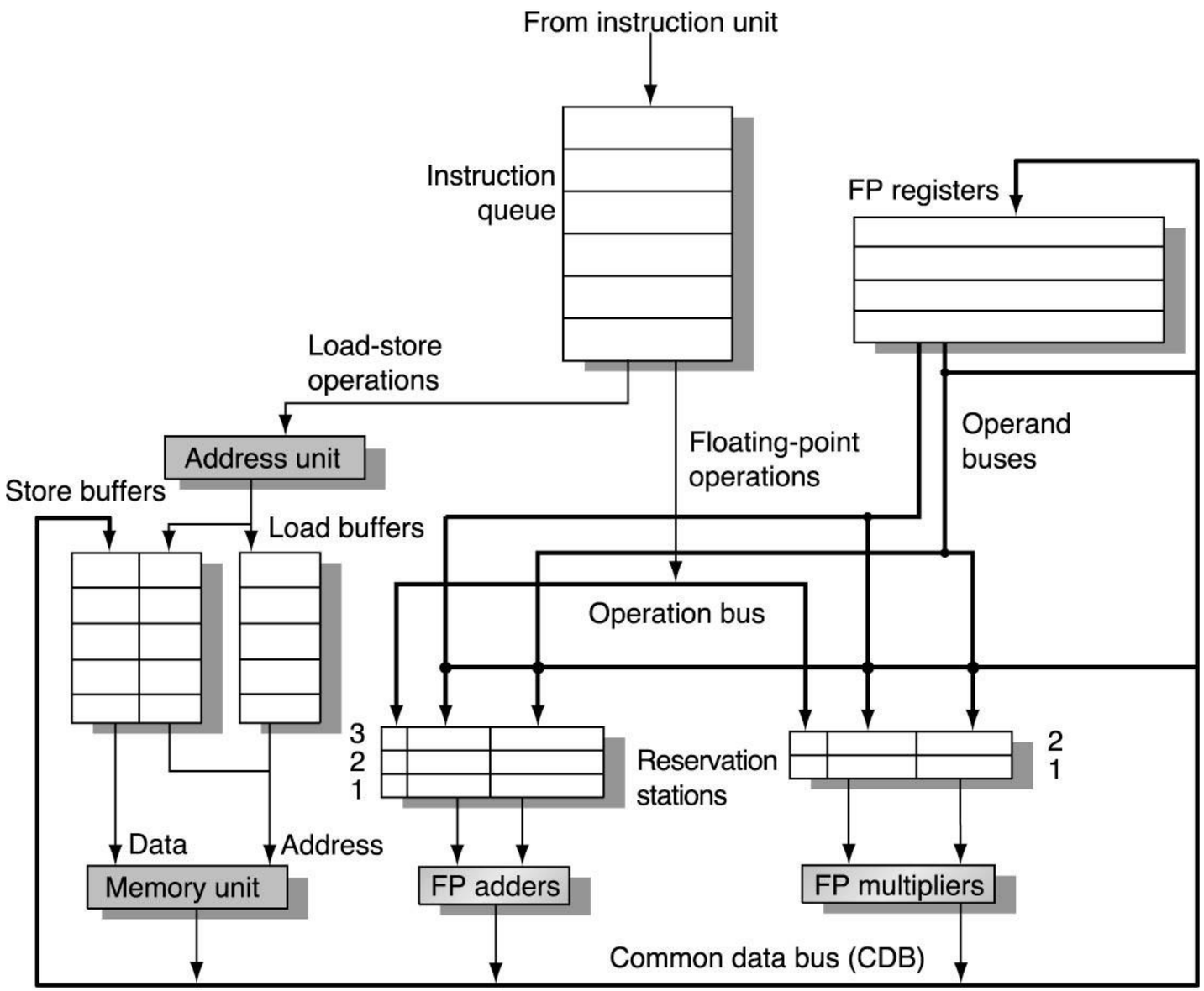
Photo: Columbia University Adminstrative Information Systems archive.

Feature of Tomasulo Alrogithm

- Control & Inst/operand buffers associated with FUs called "reservation stations"
  - instructions schedule themselves (when they decide they are ready)
- Register specifiers in instructions replaced by pointers to reserveration stations buffer
  - Reservation stations act as operand storage
- HW renaming of registers to avoid WAR and WAW hazards
  - Each register read as soon as available. When possible, read at instruction dispatch (D->reservation station)
- Otherwise read of Common Data Bus (CDB) broadcasts from and FU to all reservation stations
  - Each reservation station, register file, e\tc, responsible for capturing data off CDB
- Load and Store Queues treated as FUs as well

Tomasulo Organization

Reservation Stations, Store Buffers and FP Registers watch the CDB



Reservation Station

Op	Qi	Qk	Vj	Vk	Ri	Rk	Busy

- Op – operation to perform in the FU (e.g. add, sub, …)
- Qj, Qk – Reservation Station producing the source registers (if 0, then value captured from register file)
- Vj, Vk – Value of source operands
- Rj, Rk – Readiness of Vj and Vk
- Busy – reservation station is busy

- Table to keep track of whether we can read a valid register value from the RF at dispatch or reservation station

R0:	value	res. station id
R1:	value	res. station id
R2:	value	res. station id
R3:	value	res. station id

Three Steps of Tomasulo Algorithm

1. Dispatch (book calls this issue) – get instruction from FP Inst Queue (in order)

   if reservation station free (structural dependency), the IQ dispatches inst & sends operands (rename registers)

   update register file table – this RS is now the source of the register value

2. Execution (issue) - operate on operands (EX)

   when both operands are ready and functional unit free then execute.

   if not, watch CDB for result

3. Write result – finish execution (WB)

   write result (and reservation id) on Common Data Bus to all waiting units

   mark reservation station as free (not busy)

Limitations of Tomasulo

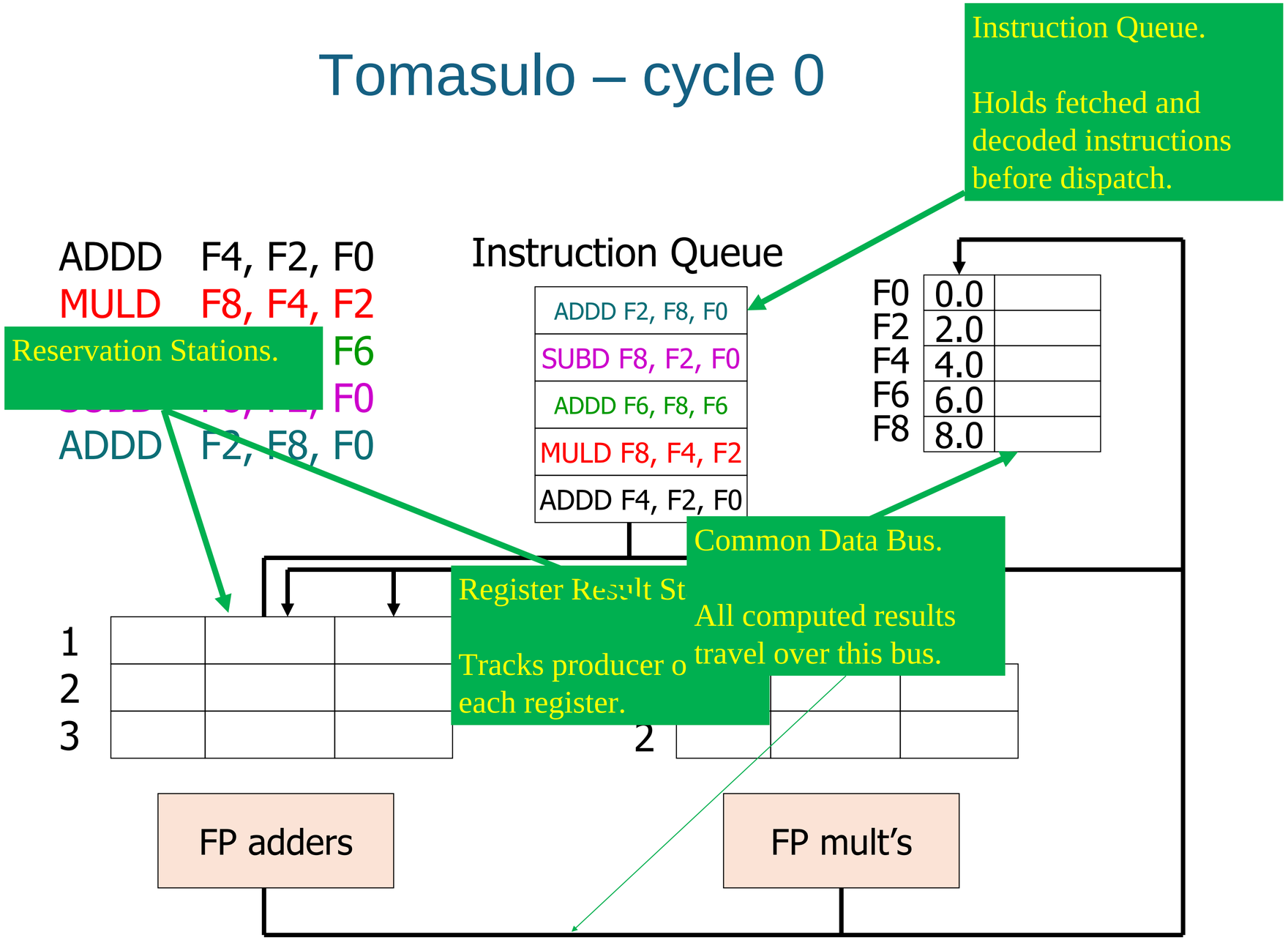
- basic algorithm was pretty limited, in particular, did not support speculative execution (can't execute past a predicted branch)

Tomasulo Example

```
ADDD  F4, F2, F0
MULD  F8, F4, F2
ADDD  F6, F8, F6
SUBD  F8, F2, F0
ADDD  F2, F8, F0
```

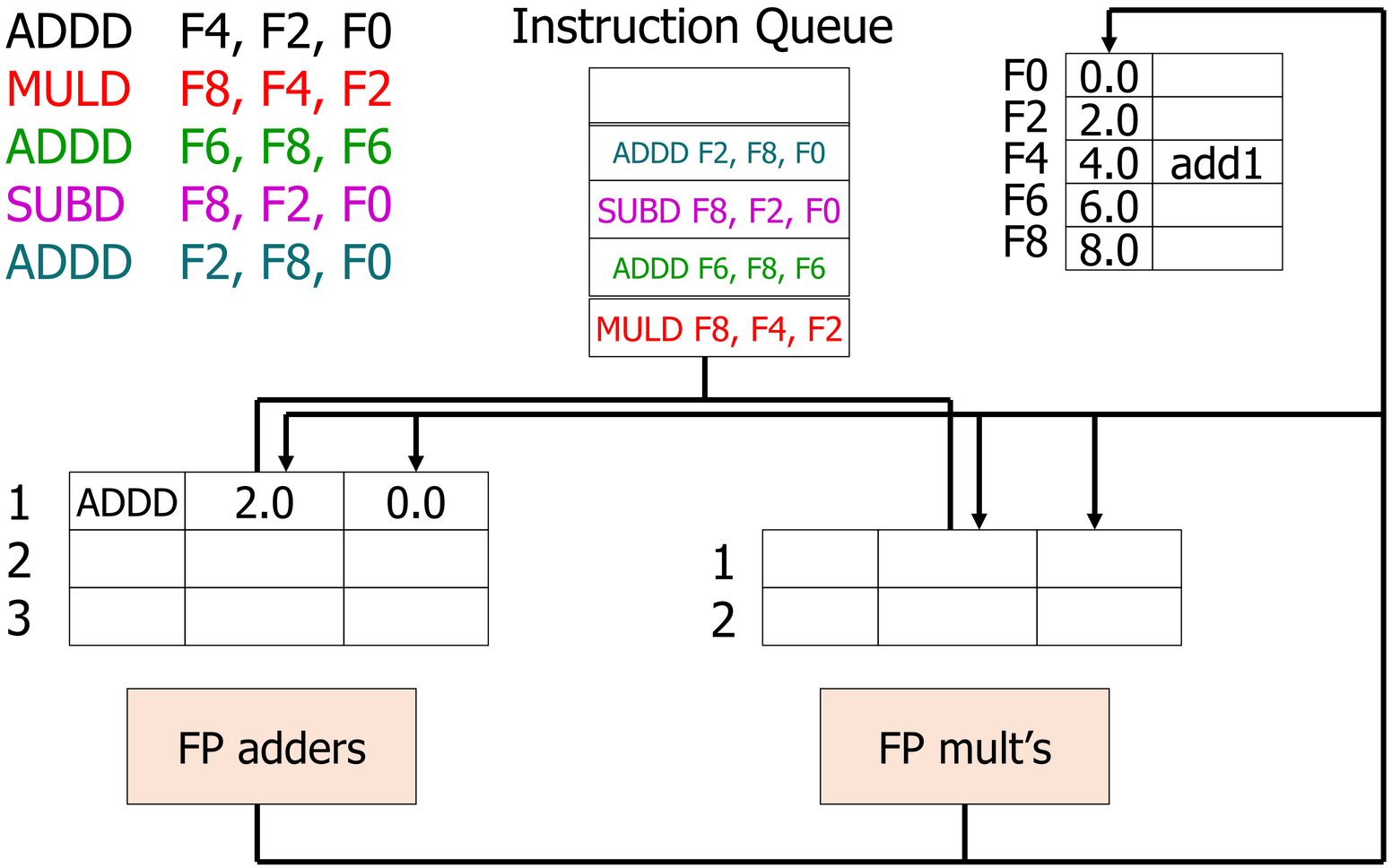
Multiply takes 10 clocks, add/sub take 4

Tomasulo – cycle 0



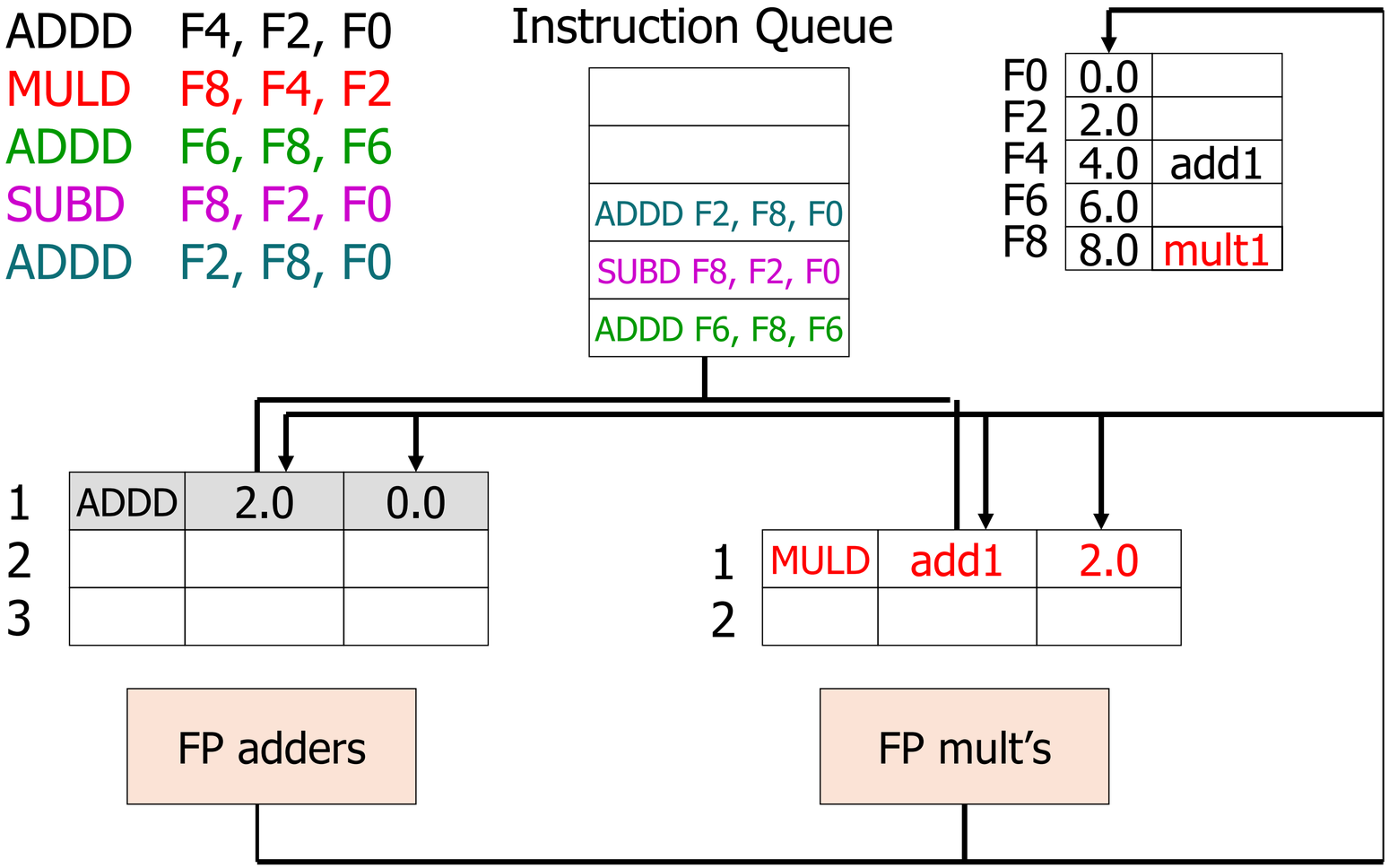
Tomasulo – cycle 1

ADDD	F4, F2, F0
MULD	F8, F4, F2
ADDD	F6, F8, F6
SUBD	F8, F2, F0
ADDD	F2, F8, F0

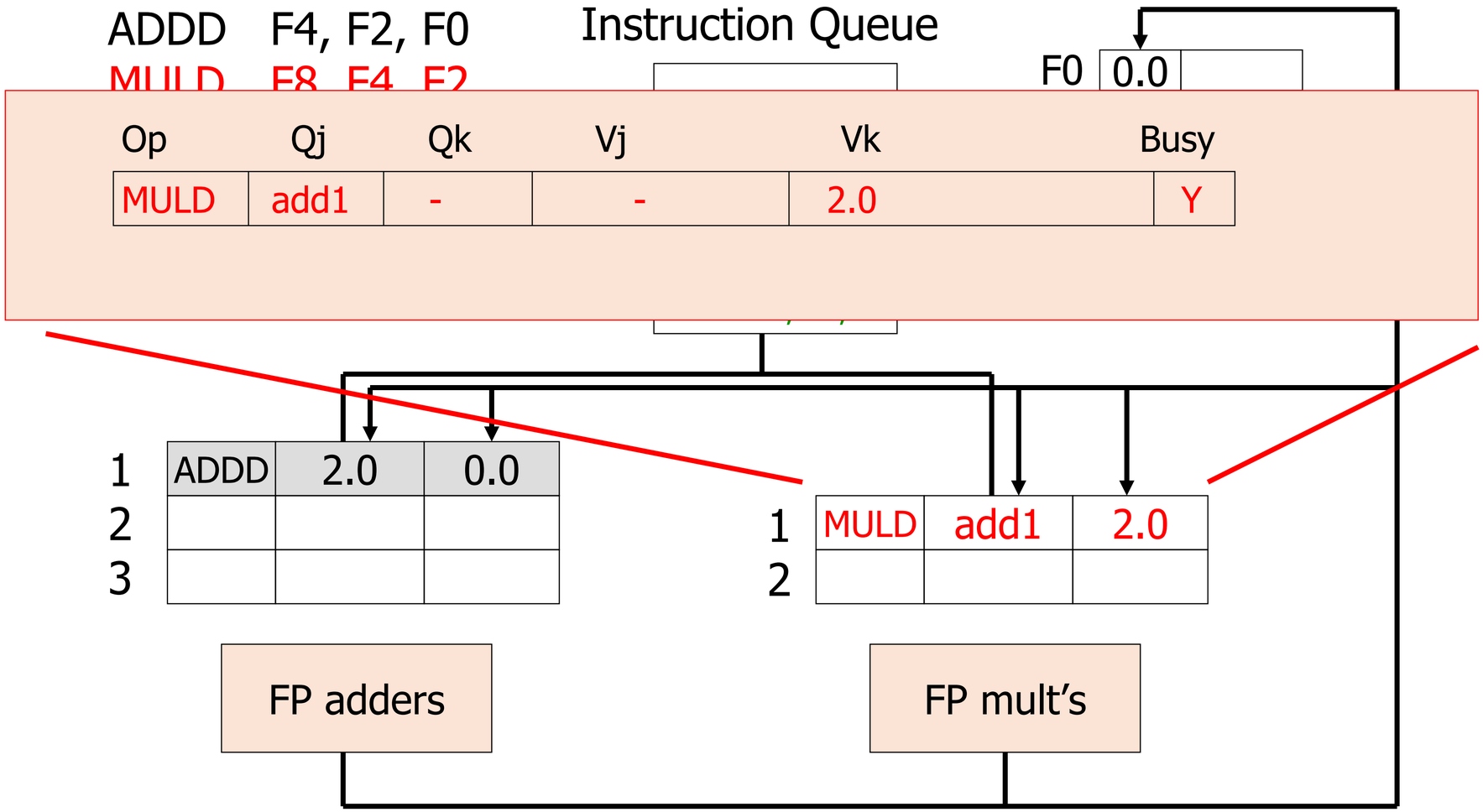


Tomasulo – cycle 2

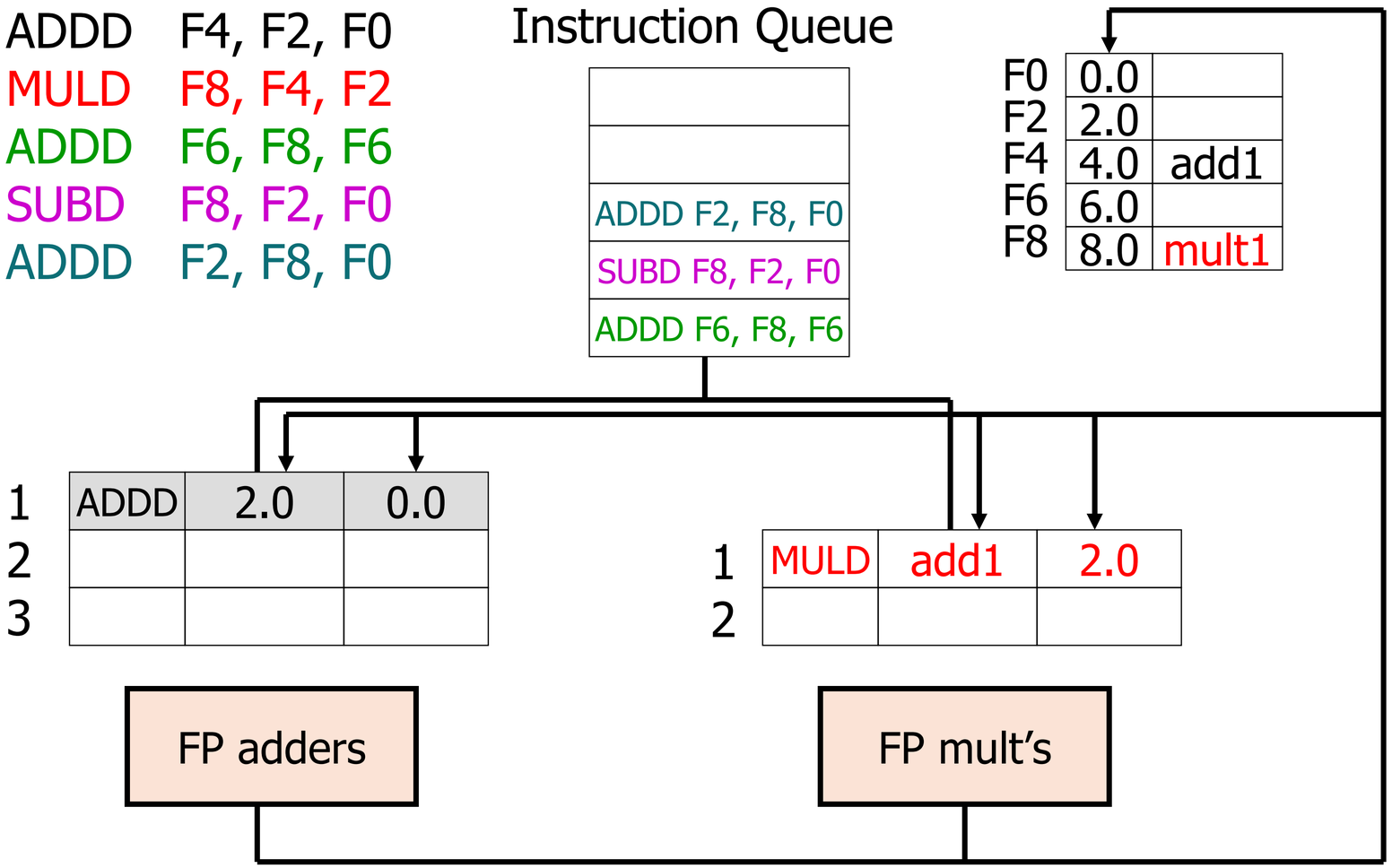
ADDD F4, F2, F0
MULD F8, F4, F2
ADDD F6, F8, F6
SUBD F8, F2, F0
ADDD F2, F8, F0



Tomasulo – cycle 2

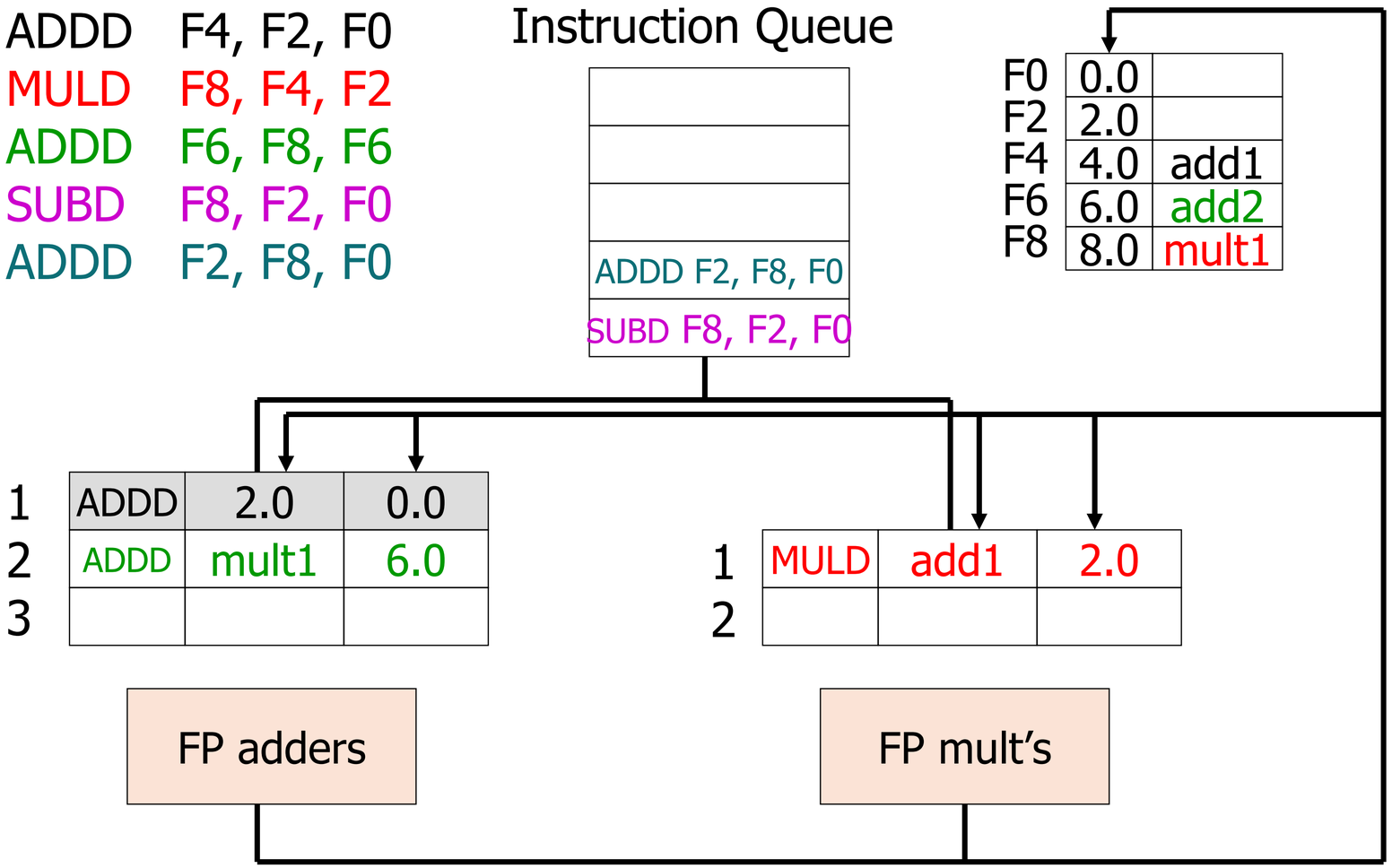


Tomasulo – cycle 2



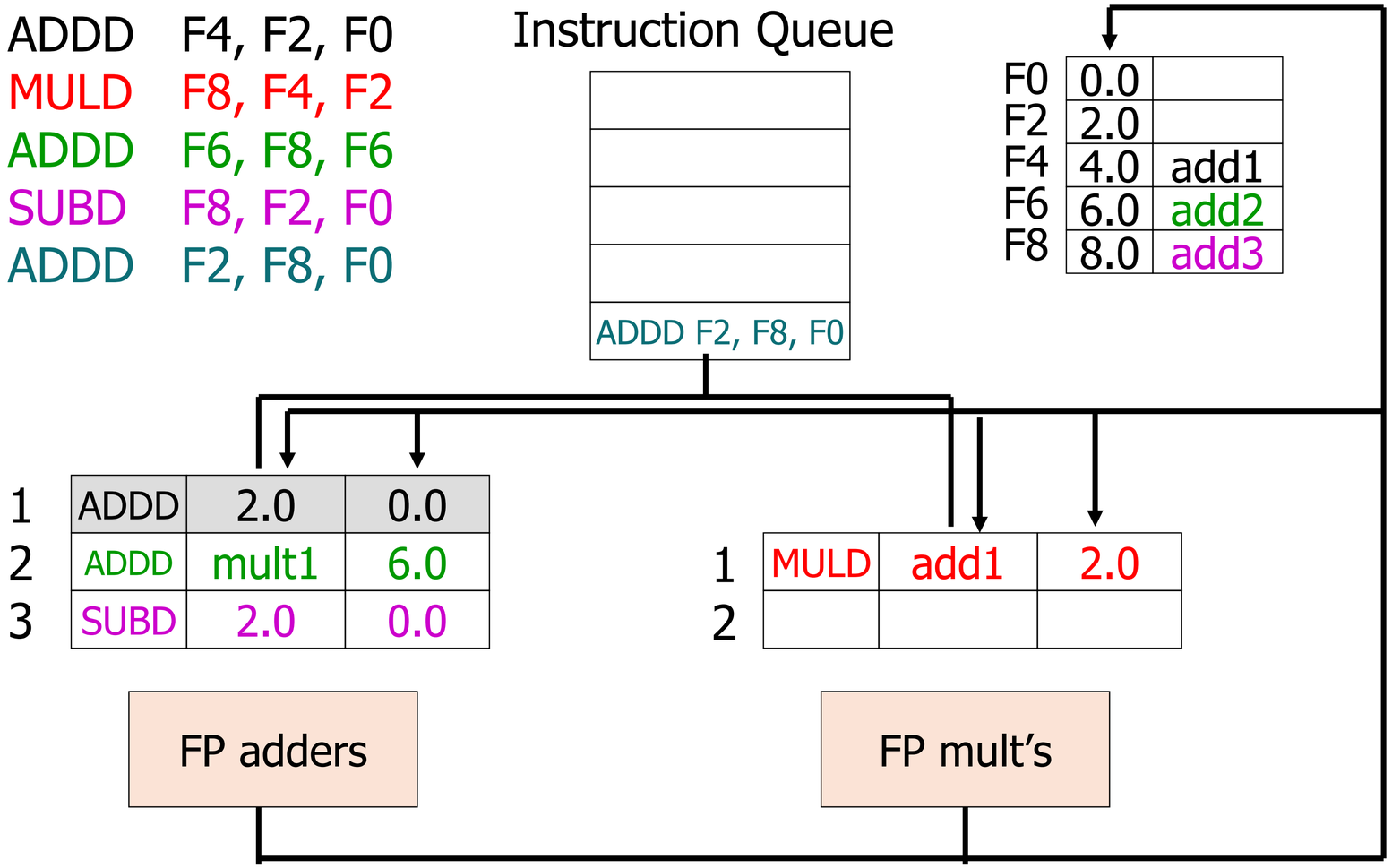
Tomasulo – cycle 3

ADDD	F4, F2, F0
MULD	F8, F4, F2
ADDD	F6, F8, F6
SUBD	F8, F2, F0
ADDD	F2, F8, F0



Tomasulo – cycle 4

ADDD	F4, F2, F0
MULD	F8, F4, F2
ADDD	F6, F8, F6
SUBD	F8, F2, F0
ADDD	F2, F8, F0



Tomasulo – cycle 5

ADDD	F4, F2, F0
MULD	F8, F4, F2
ADDD	F6, F8, F6
SUBD	F8, F2, F0
ADDD	F2, F8, F0

Instruction Queue

ADDD F2, F8, F0

F0	0.0	
F2	2.0	
F4	***2.0***	-
F6	6.0	add2
F8	8.0	add3

1	ADDD	2.0	0.0
2	ADDD	mult1	6.0
3	SUBD	2.0	0.0

1	MULD	***2.0***	2.0
2			

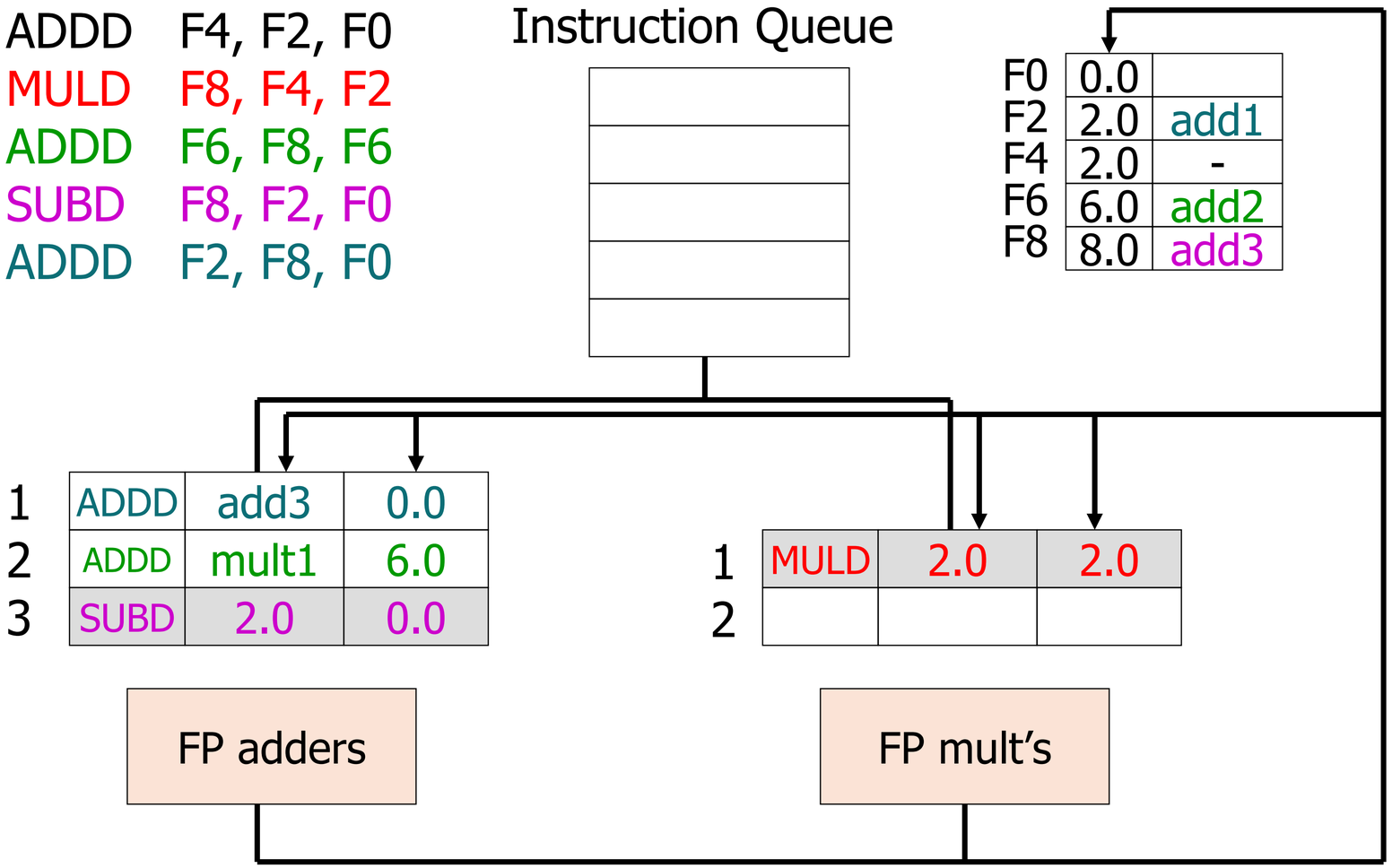
FP adders

FP mult's

2.0 (add1 result)

Tomasulo – cycle 6

ADDD F4, F2, F0
MULD F8, F4, F2
ADDD F6, F8, F6
SUBD F8, F2, F0
ADDD F2, F8, F0



Tomasulo – cycle 8

ADDD	F4, F2, F0
MULD	F8, F4, F2
ADDD	F6, F8, F6
SUBD	F8, F2, F0
ADDD	F2, F8, F0

Instruction Queue

F0	0.0	
F2	2.0	add1
F4	2.0	-
F6	6.0	add2
F8	***2.0***	-

1	ADDD	***2.0***	0.0
2	ADDD	mult1	6.0
3	SUBD	2.0	0.0

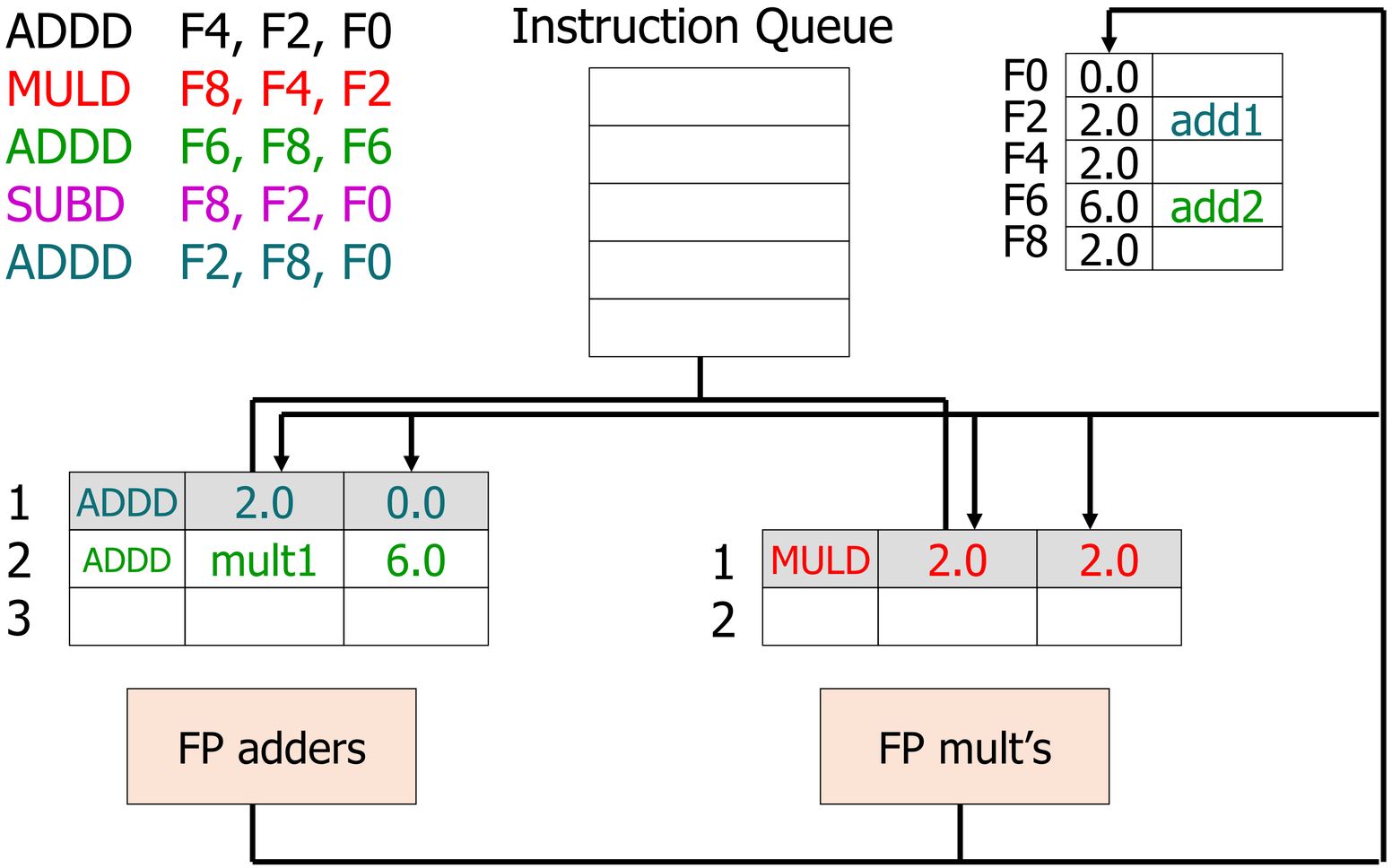
1	MULD	2.0	2.0
2			

FP adders

FP mult's

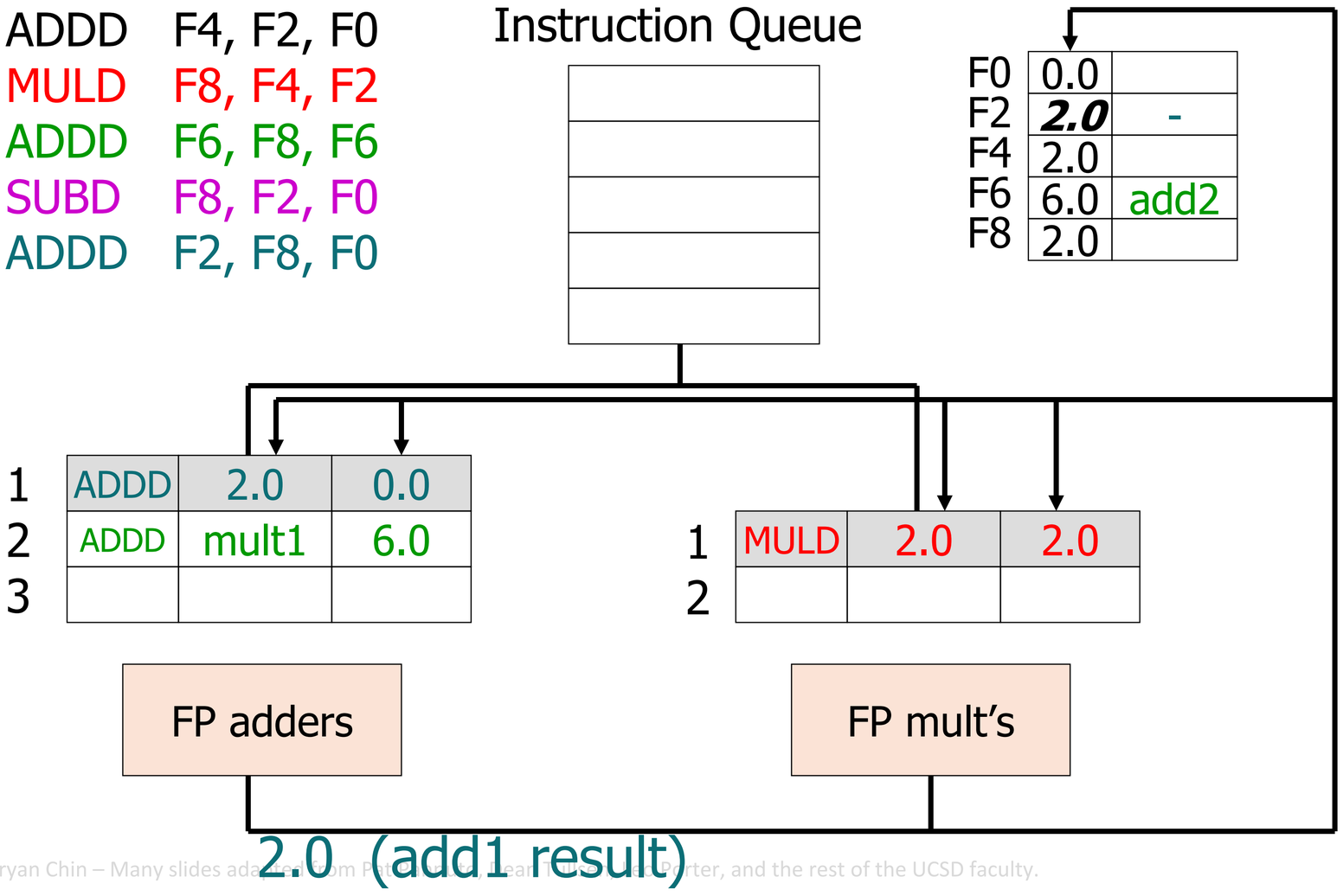
2.0 (add3 - SUBD instruction result)

Tomasulo – cycle 9



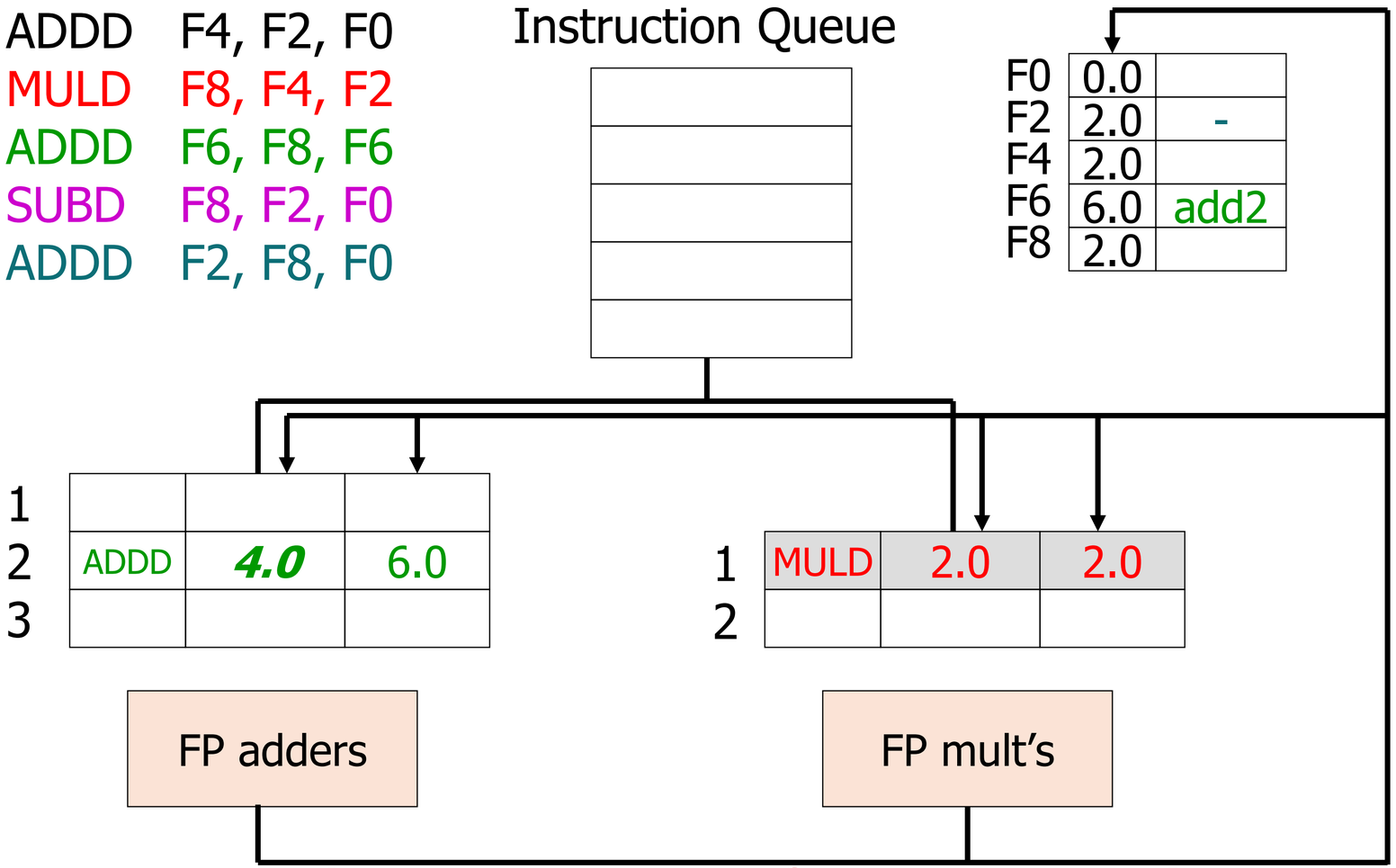
Tomasulo – cycle 12

ADDD F4, F2, F0
MULD F8, F4, F2
ADDD F6, F8, F6
SUBD F8, F2, F0
ADDD F2, F8, F0



Tomasulo – cycle 15

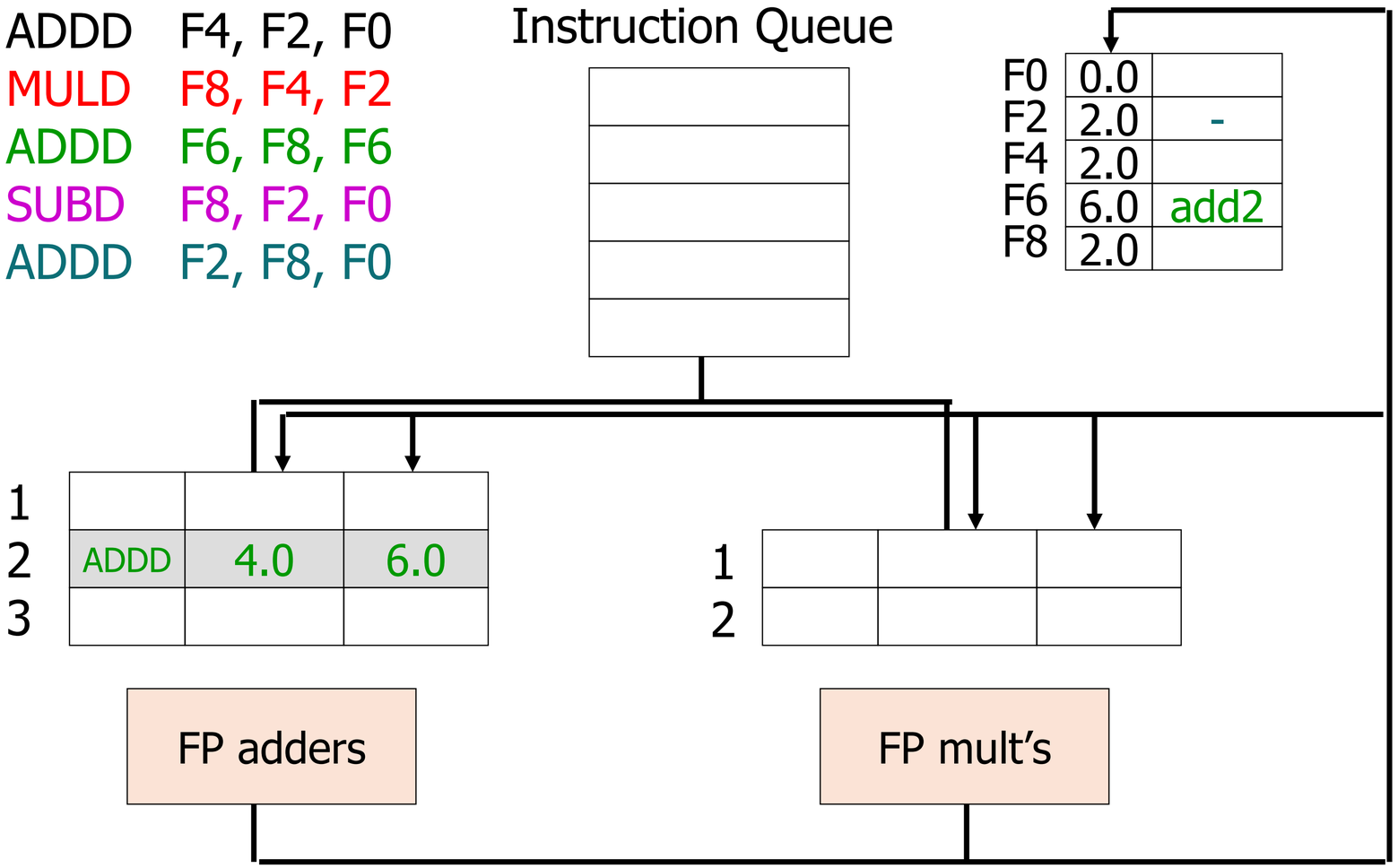
ADDD F4, F2, F0
MULD F8, F4, F2
ADDD F6, F8, F6
SUBD F8, F2, F0
ADDD F2, F8, F0



4.0 (mult1 result – forward to add2, but doesn't write RF)

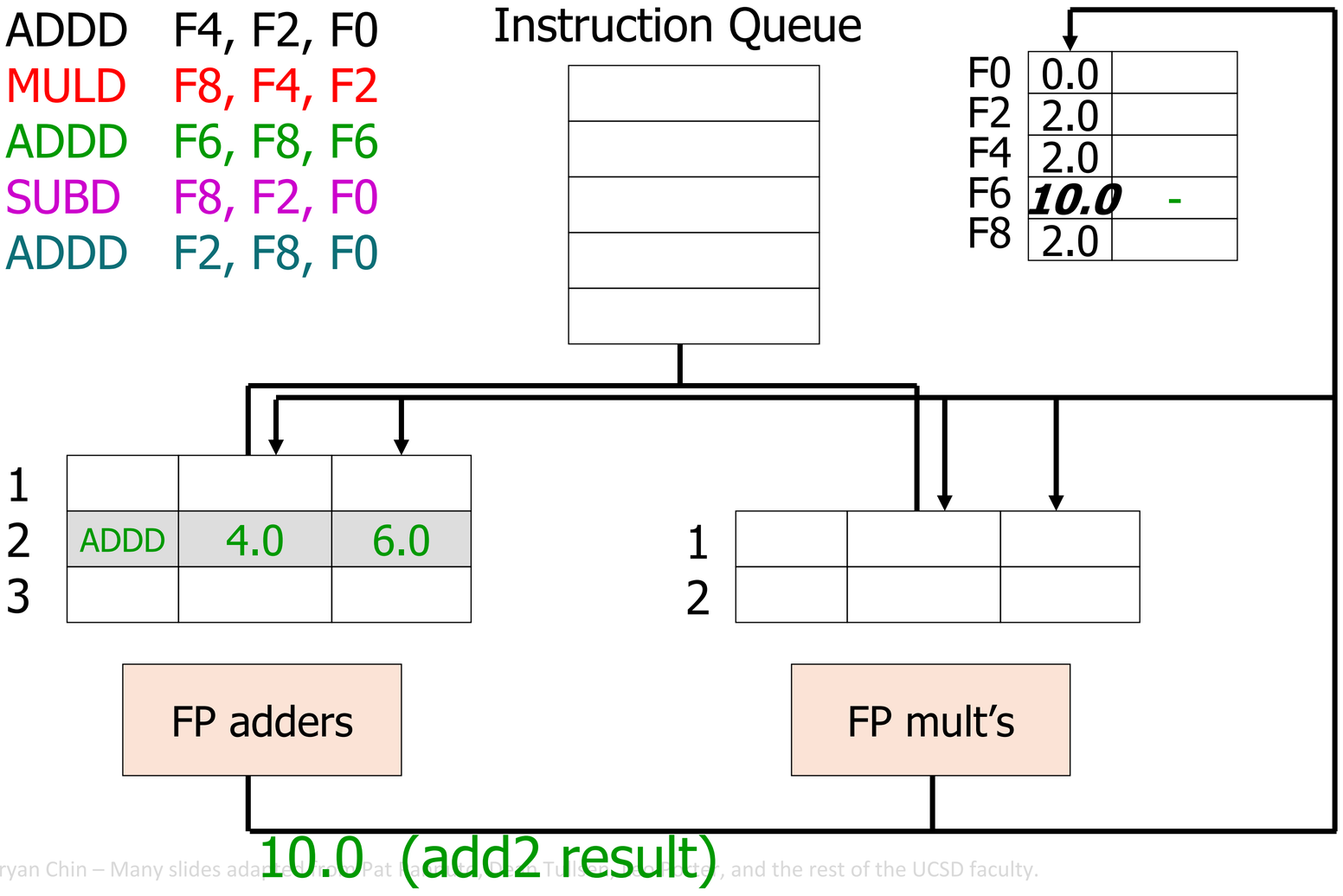
Tomasulo – cycle 16

ADDD F4, F2, F0
MULD F8, F4, F2
ADDD F6, F8, F6
SUBD F8, F2, F0
ADDD F2, F8, F0



Tomasulo – cycle 19

ADDD F4, F2, F0
MULD F8, F4, F2
ADDD F6, F8, F6
SUBD F8, F2, F0
ADDD F2, F8, F0



Tomasulo Summary

- Prevents Register File being a bottleneck
  - reservation stations add more storage for more instances of each register
- Avoids WAR and WAW hazards of scoreboard
  - register renaming as a side effect (logical register names replaced by reservation station pointers)
- Dynamic Scheduling of Instructions
- Limited to within a basic block (we will address this next)

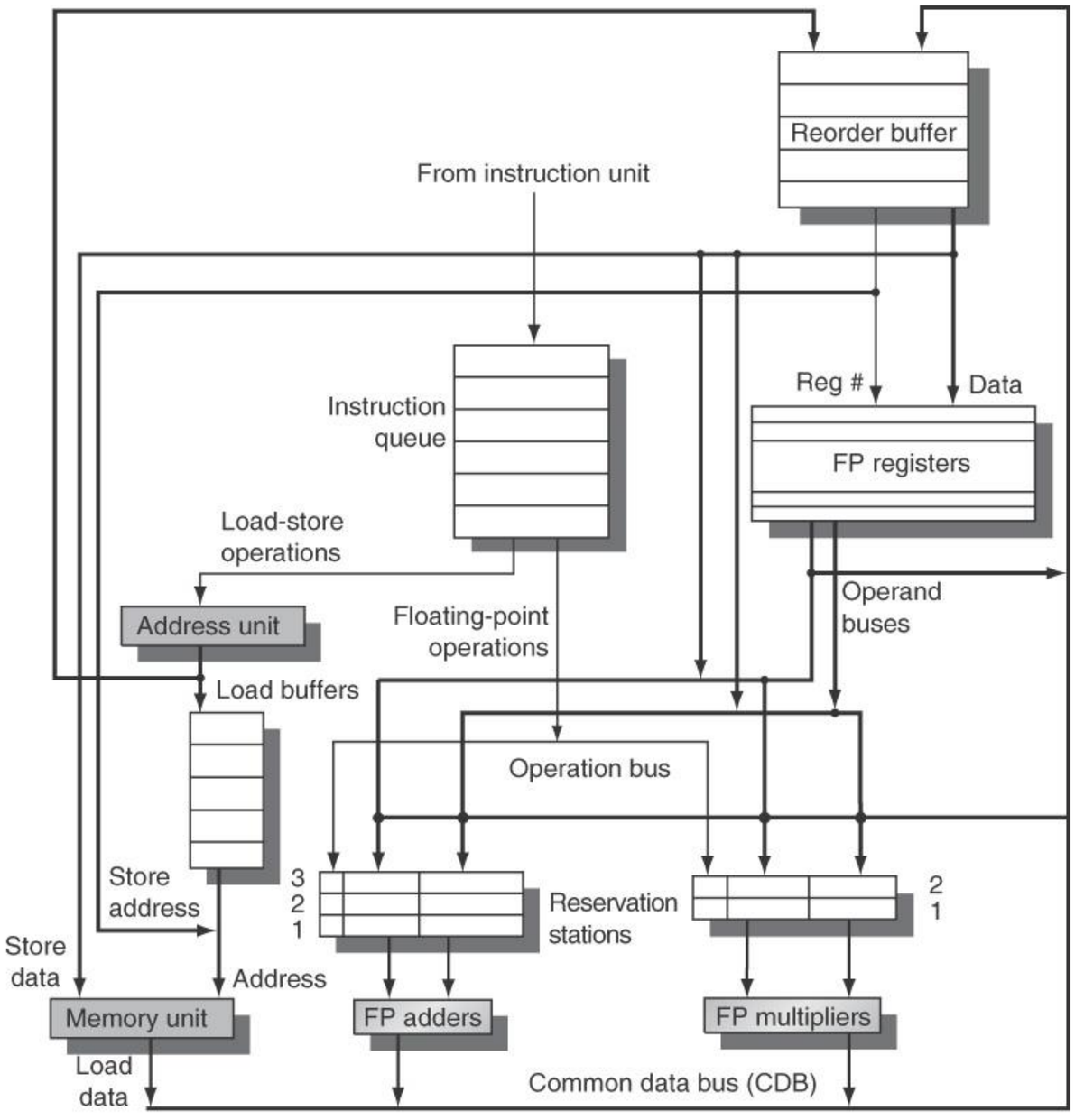
HW support for More ILP

- Speculation – allow an instruction to issue that is dependent on a branch, *without* any consequences (including exceptions) if branch is predicted incorrectly (HW undo)
- Usually combined with Dynamic Scheduling
- Tomasulo : allow *speculative* bypassing of results
  - when instructio
  - when is no longer speculative, write results (instruction *commit* or *retire*)
  - execute out of order but commit n order
  - Need some kind of intermediate storage for results (after EX but before commit)

Hardware Speculative Execution

HW Buffer for results of uncommitted instructions: reorder buffer

- can operate as operand src
- once committed results are in RF (architectural state)
- 3 fields: instr, reg addr, dest value
- ROB index instead of reservation name
- instructions commit in-order
- Easy to undo speculated instructions on mispredicted branch or exception (flush the ROB)



Four Steps of Speculative Tomasulo Algorithm

1. Dispatch– get instruction from FP Ins Queue (in order)

   If reservation station and reorder buffer slot free, dispatch instr & send operands & reorder buffer no. for destination. Operands may be read from register file or reorder buffer.

2. Execution (issue) - operate on operands (EX)

   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute

3. Write result – finish execution (WB)

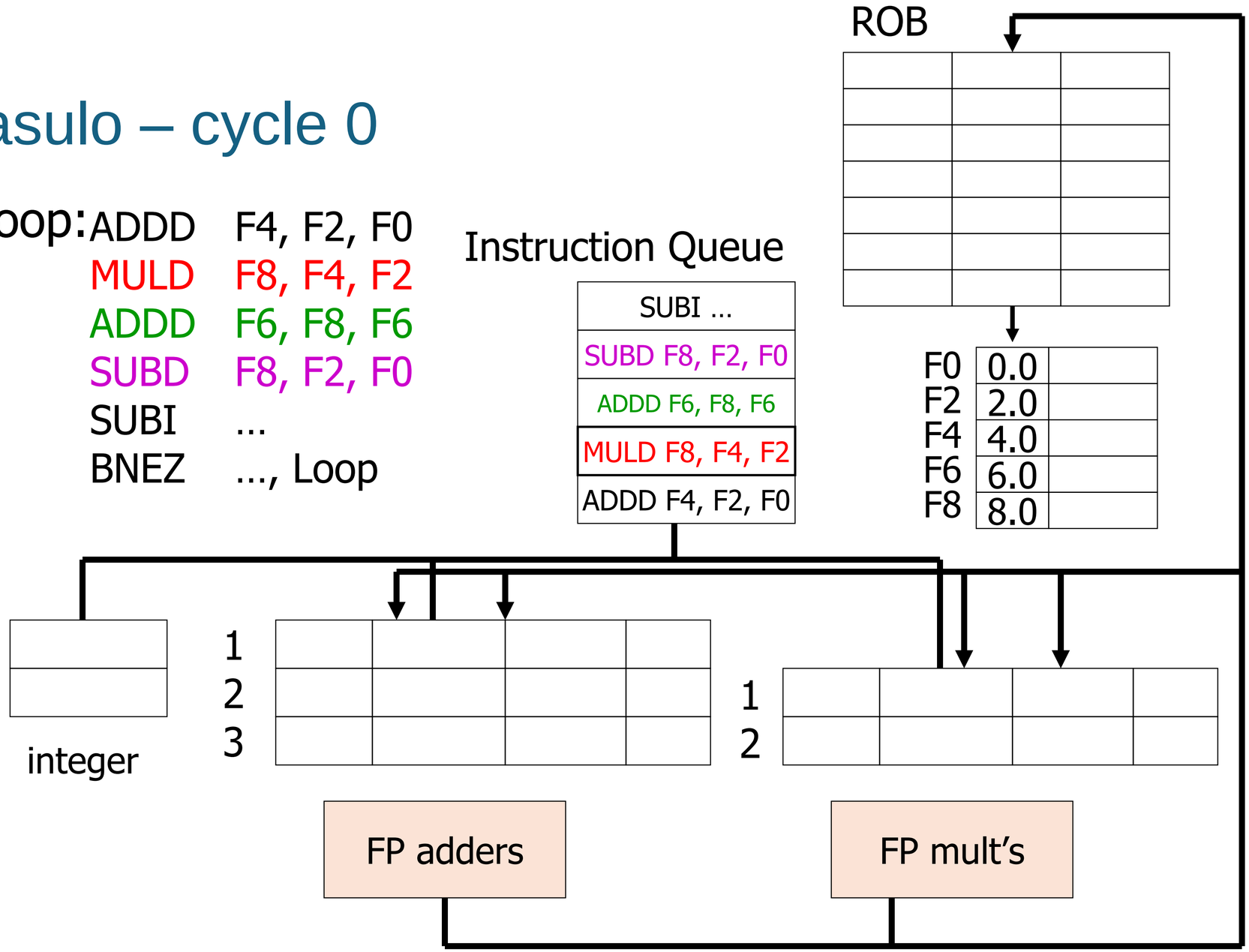
   Write on Common Data Bus to all waiting FUs & reorder buffer; mark reservation station available.

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.

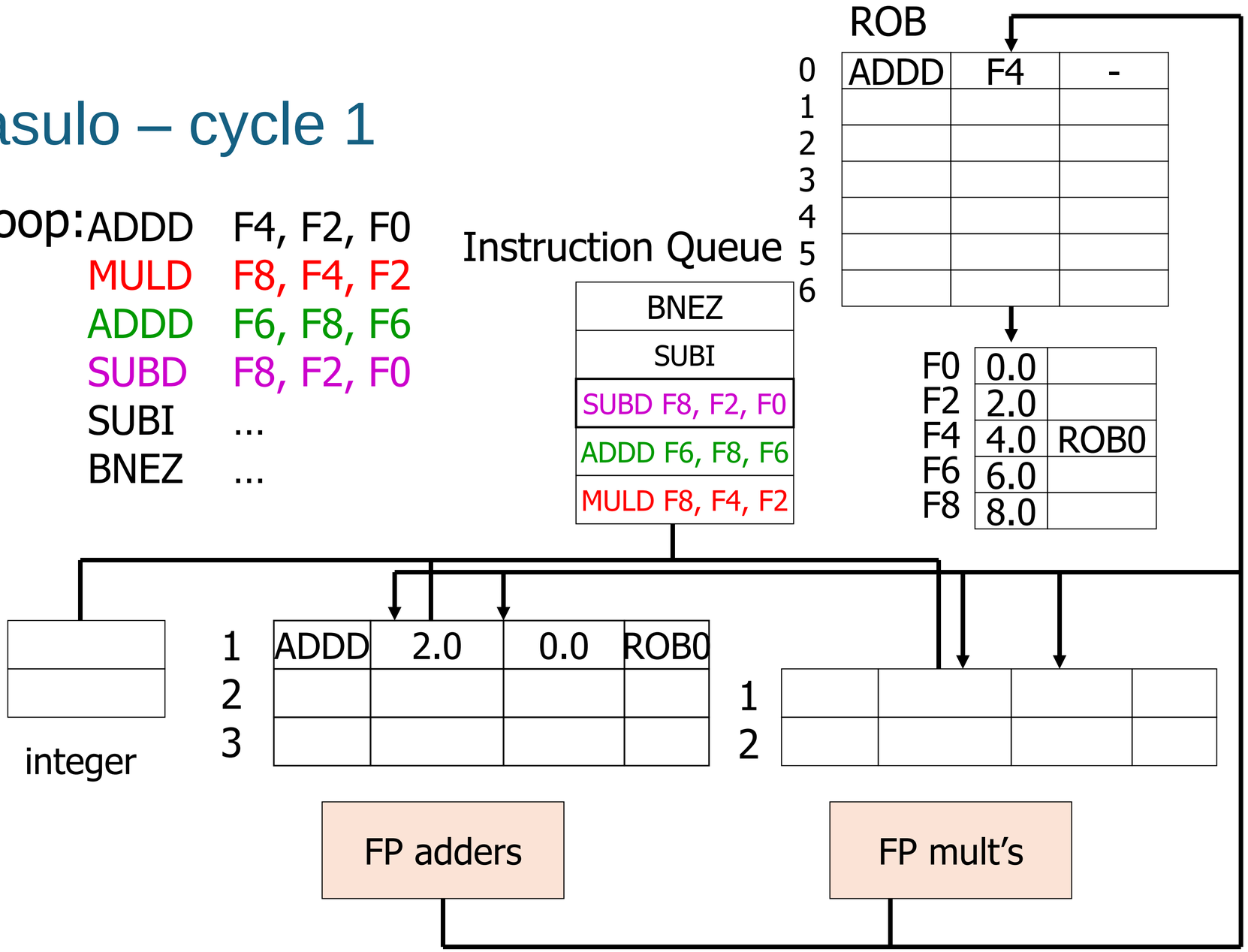
Tomasulo – cycle 0

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ..., Loop
```



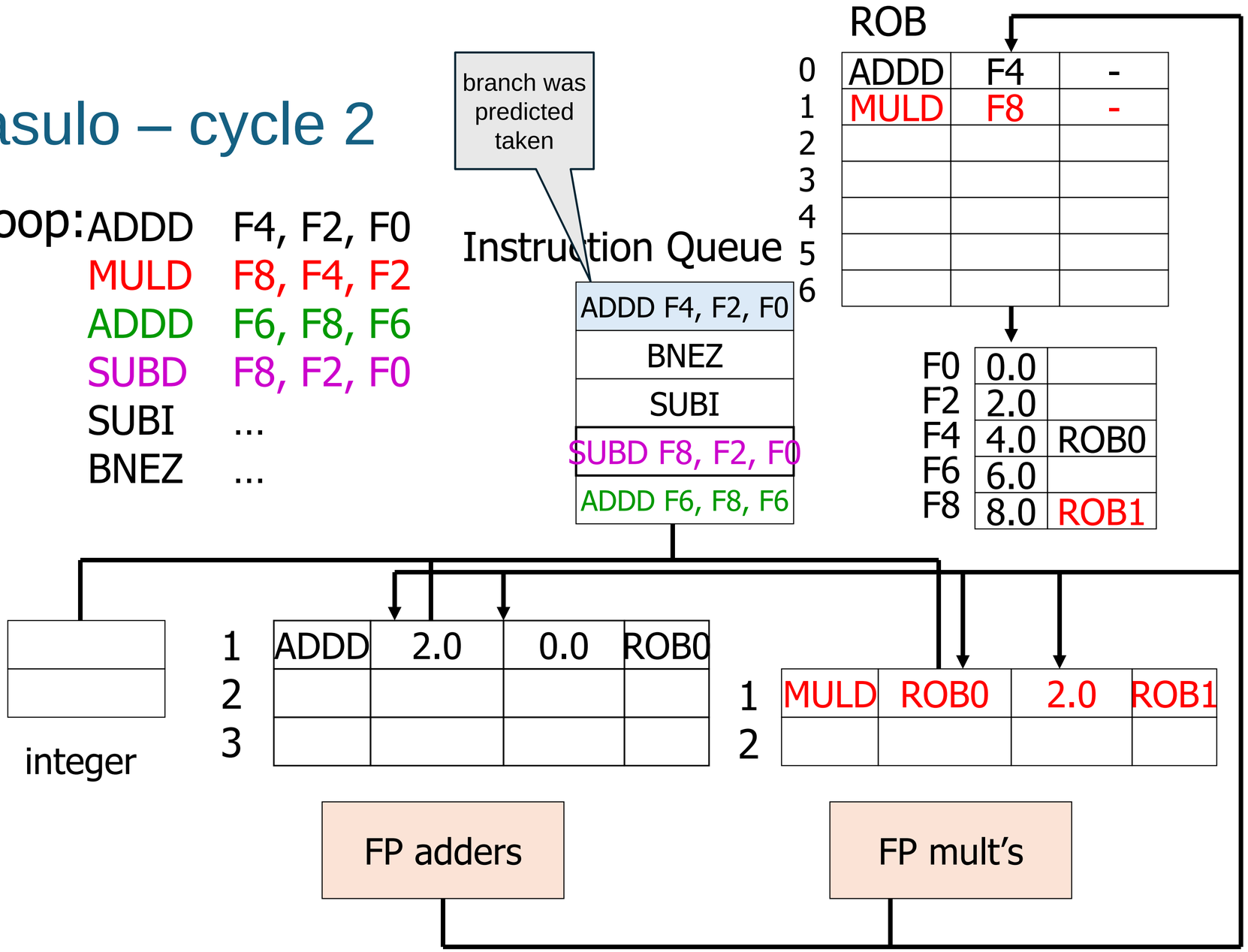
Tomasulo – cycle 1

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



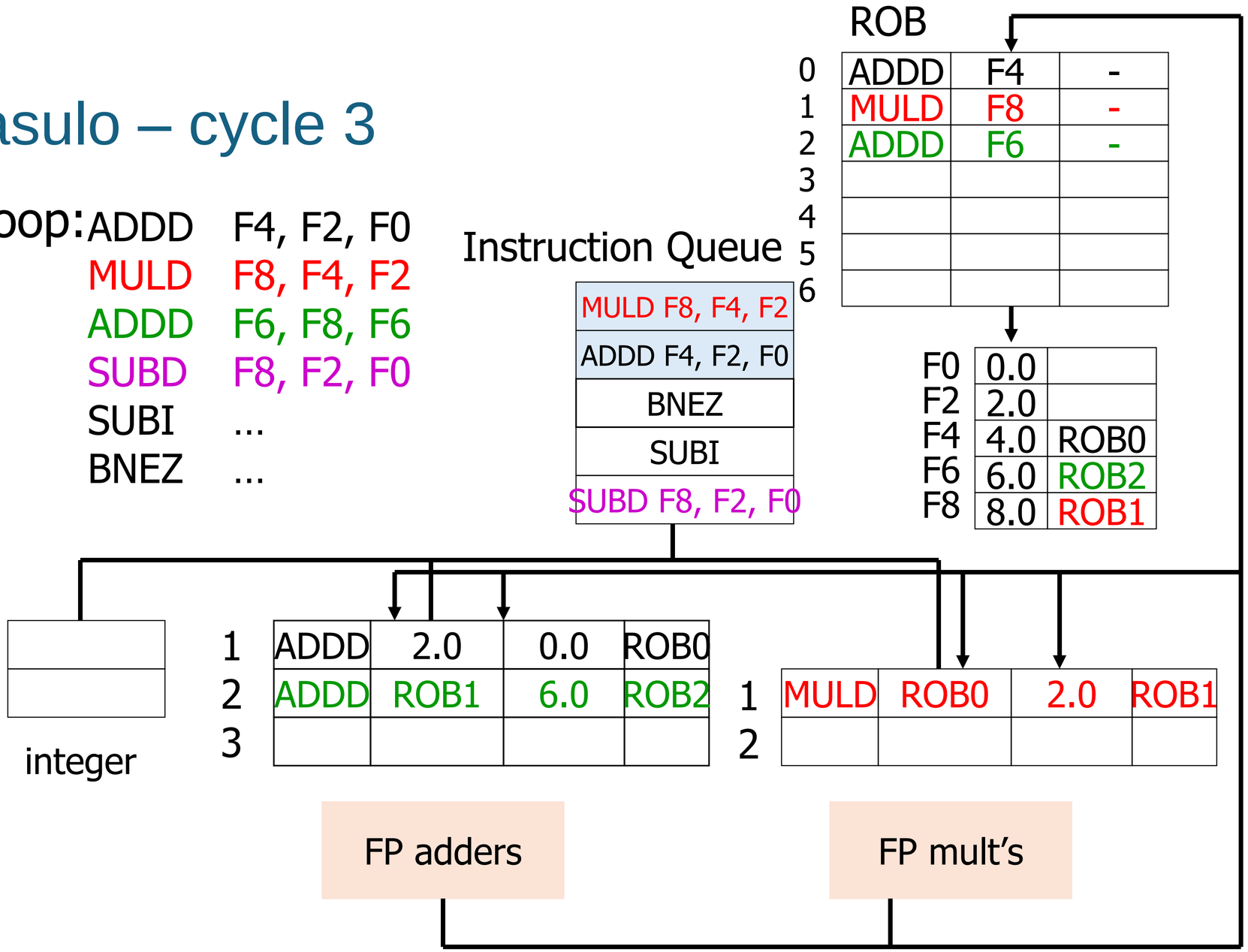
Tomasulo – cycle 2

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



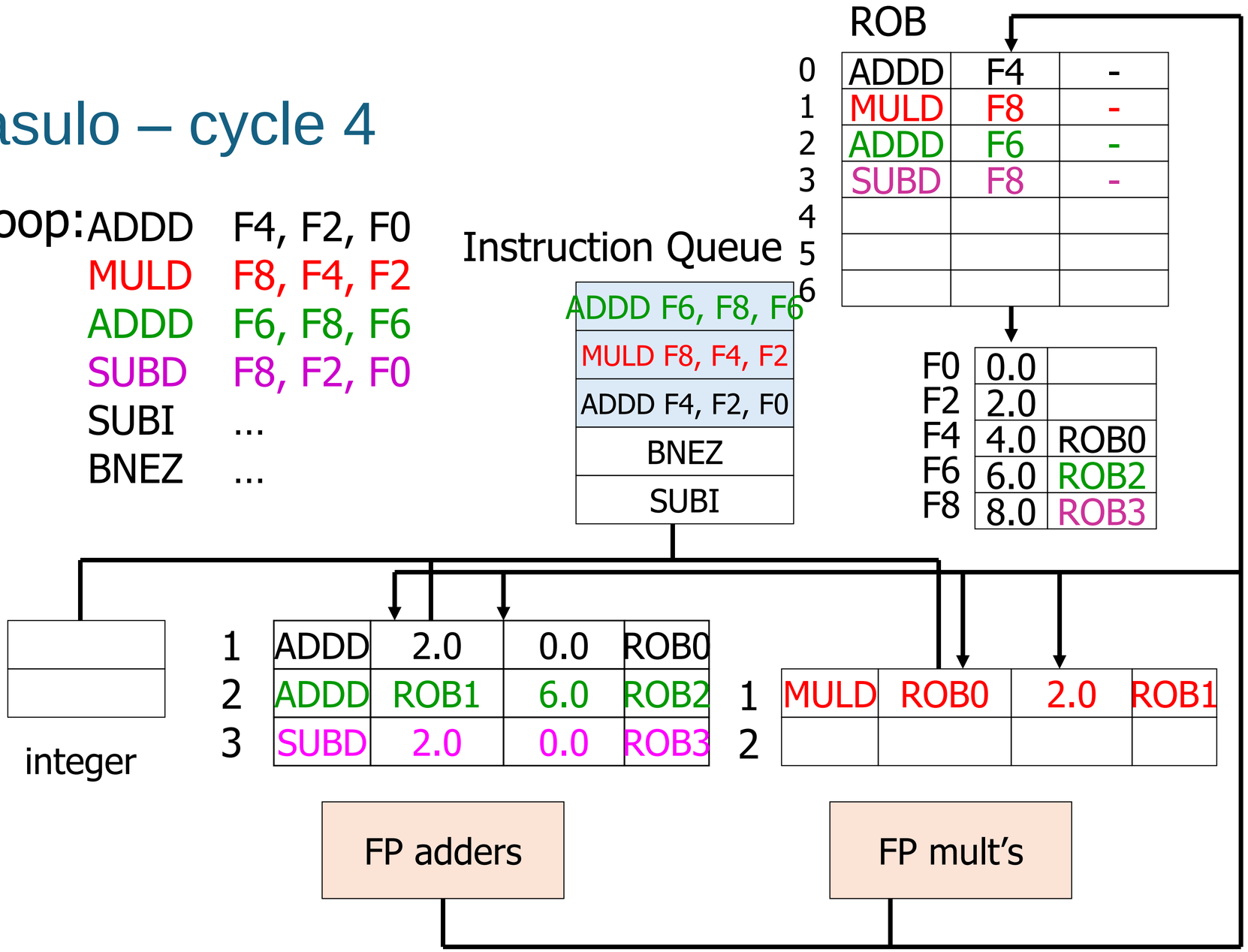
Tomasulo – cycle 3

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



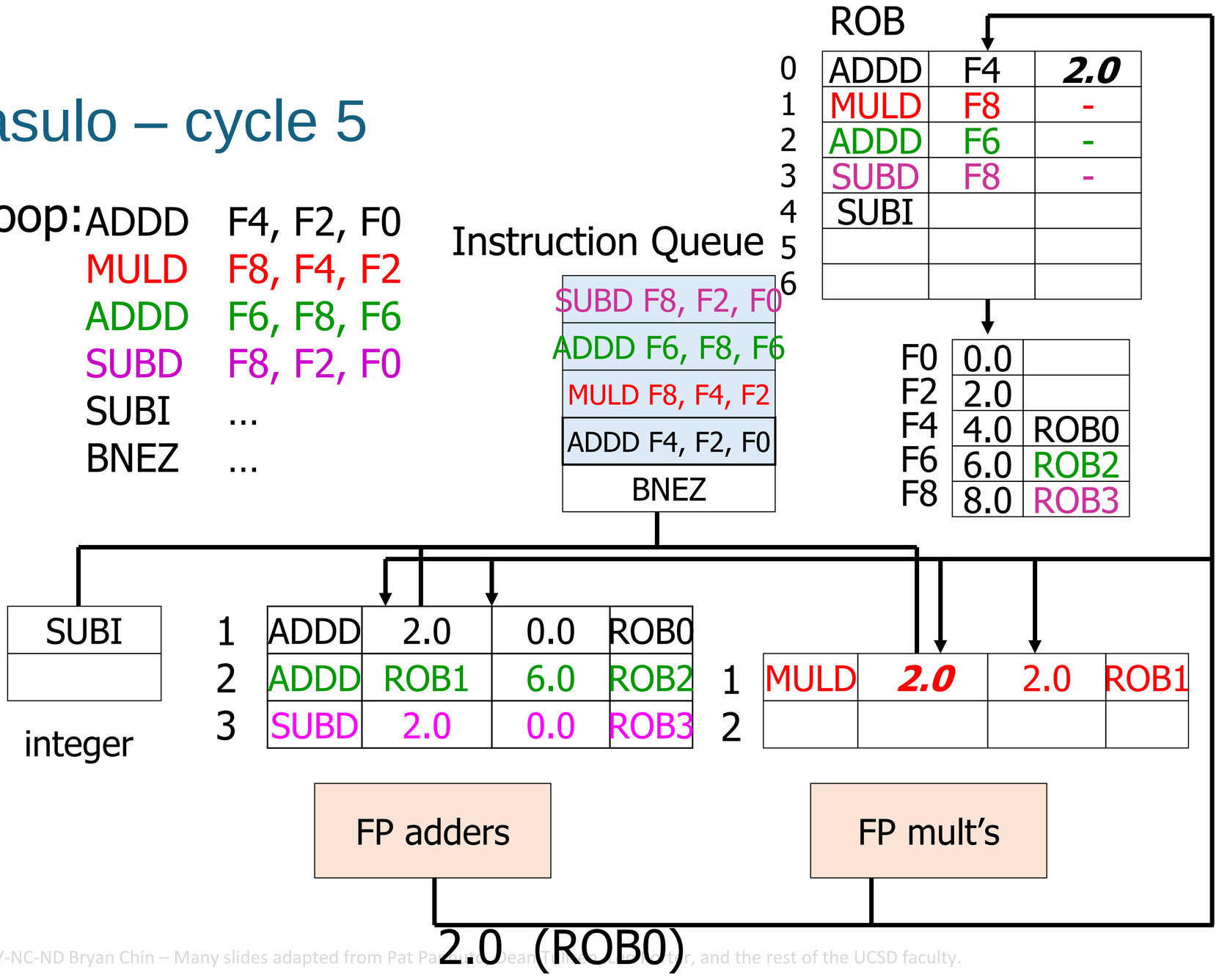
Tomasulo – cycle 4

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



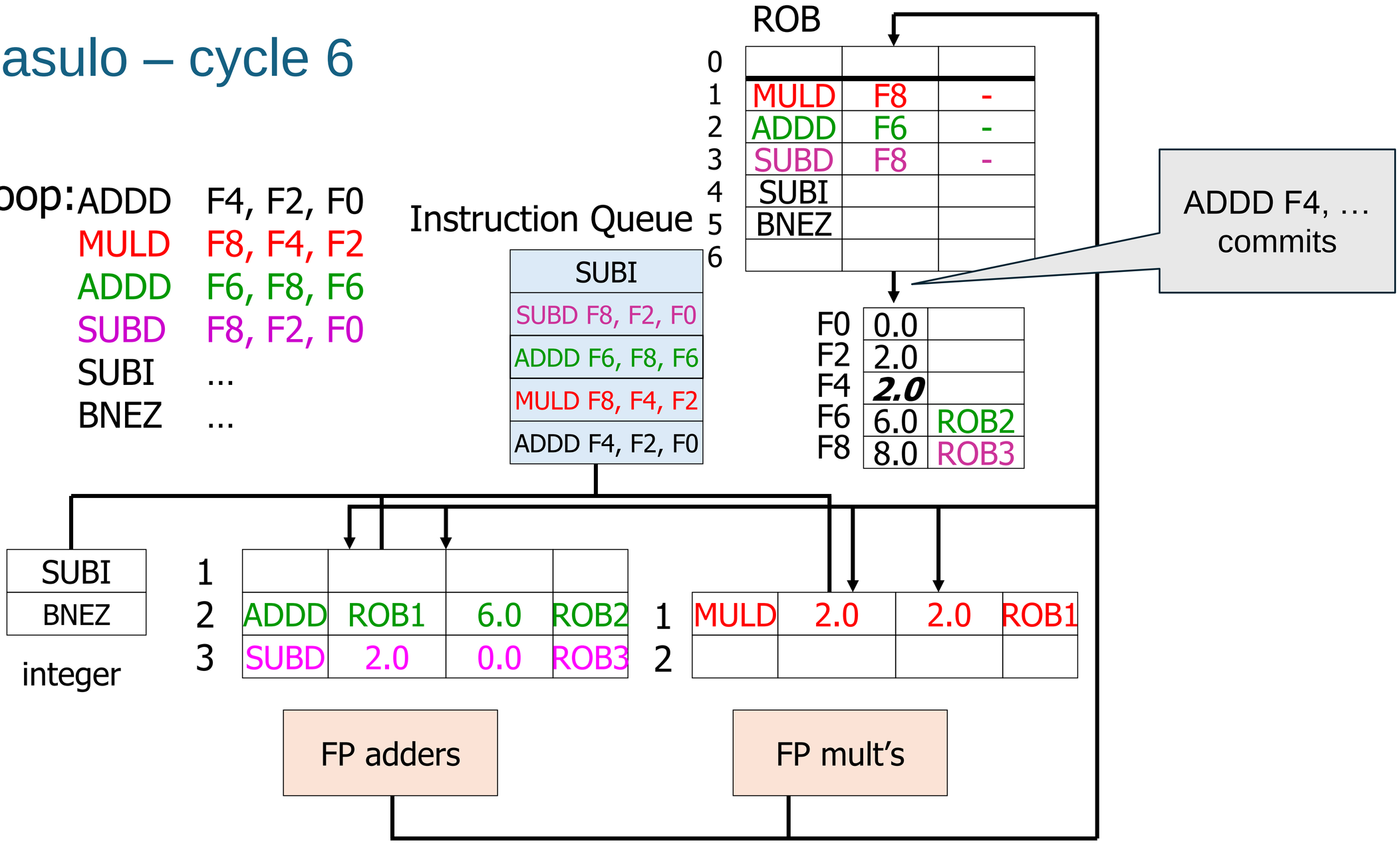
Tomasulo – cycle 5

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



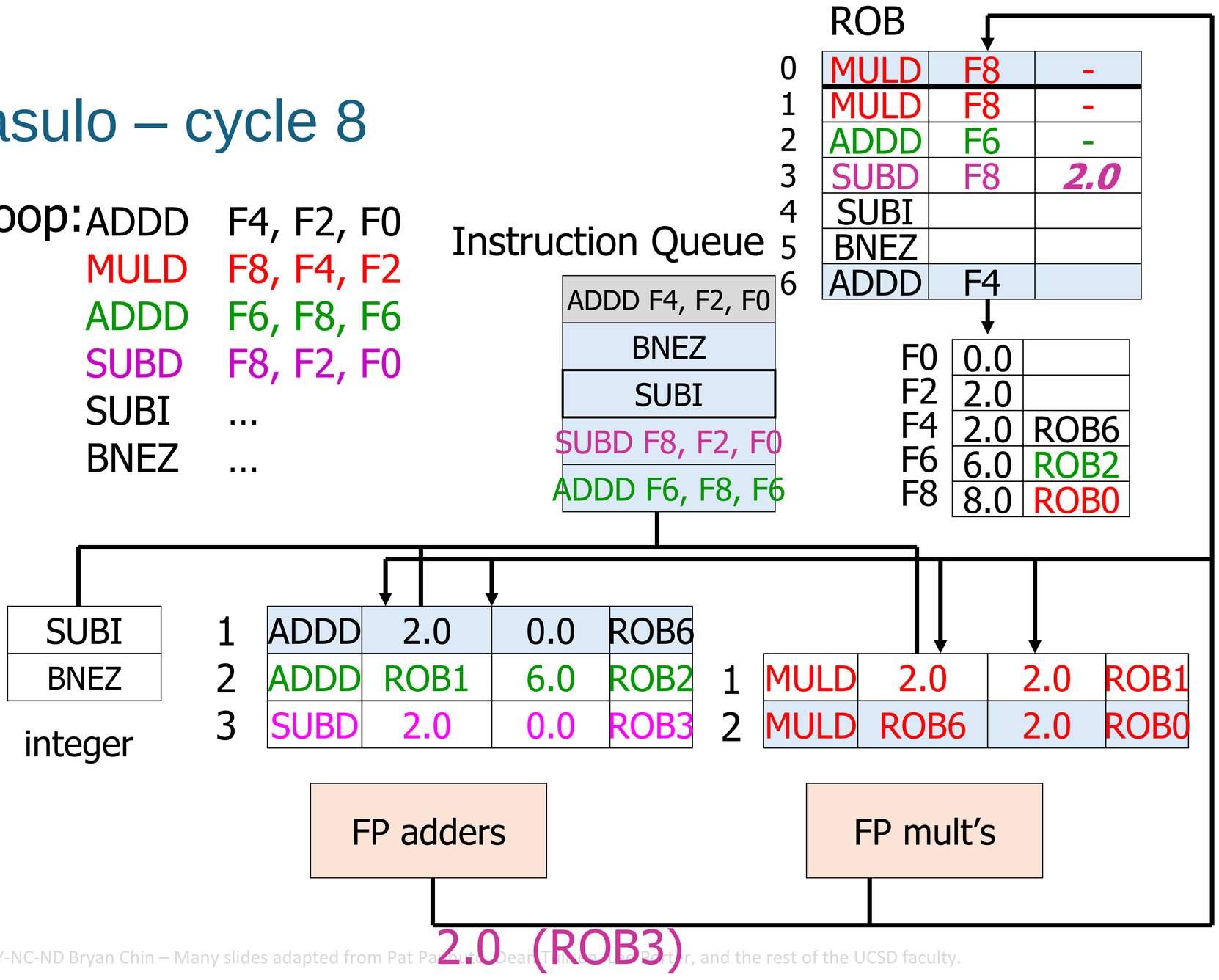
Tomasulo – cycle 6

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



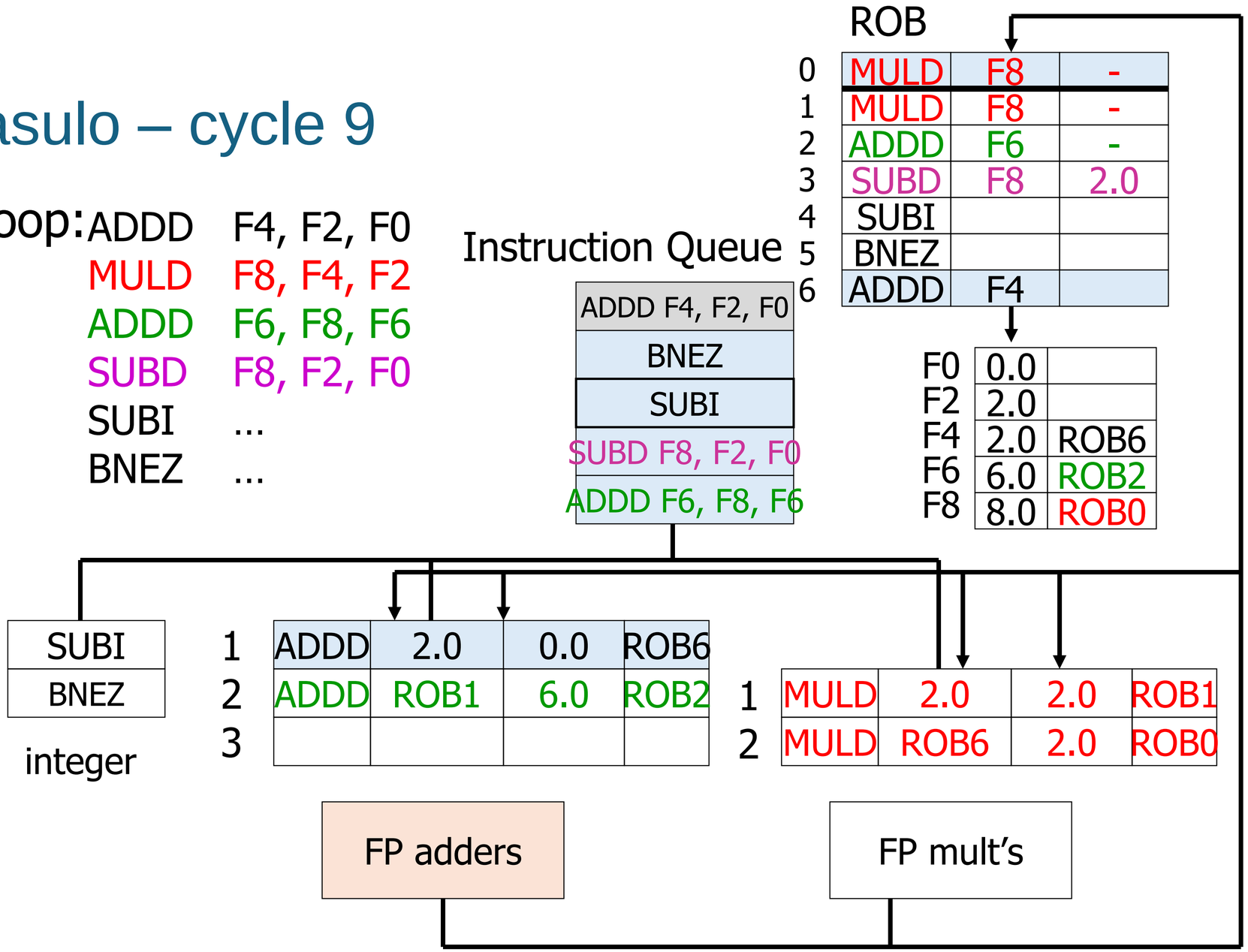
Tomasulo – cycle 8

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



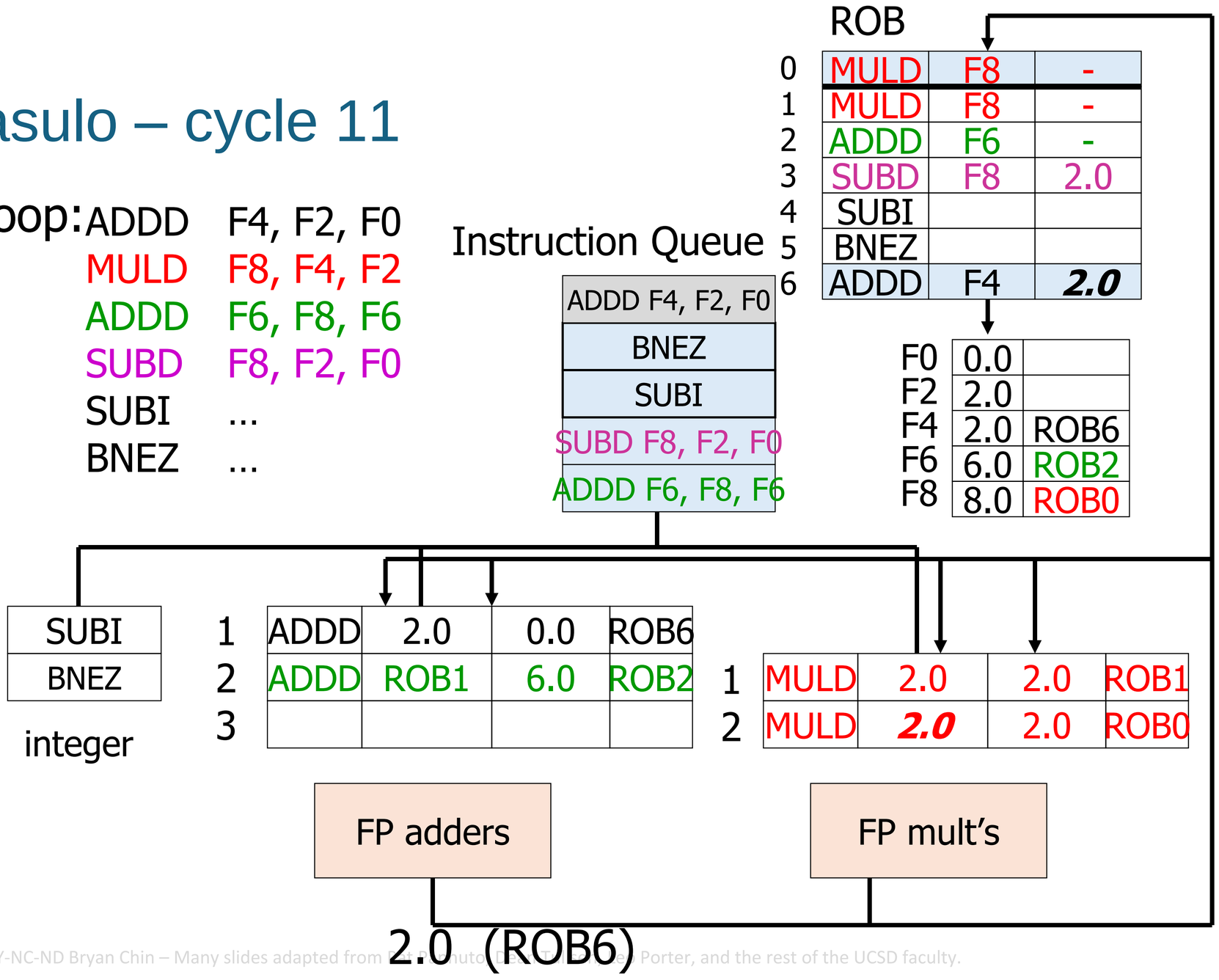
Tomasulo – cycle 9

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



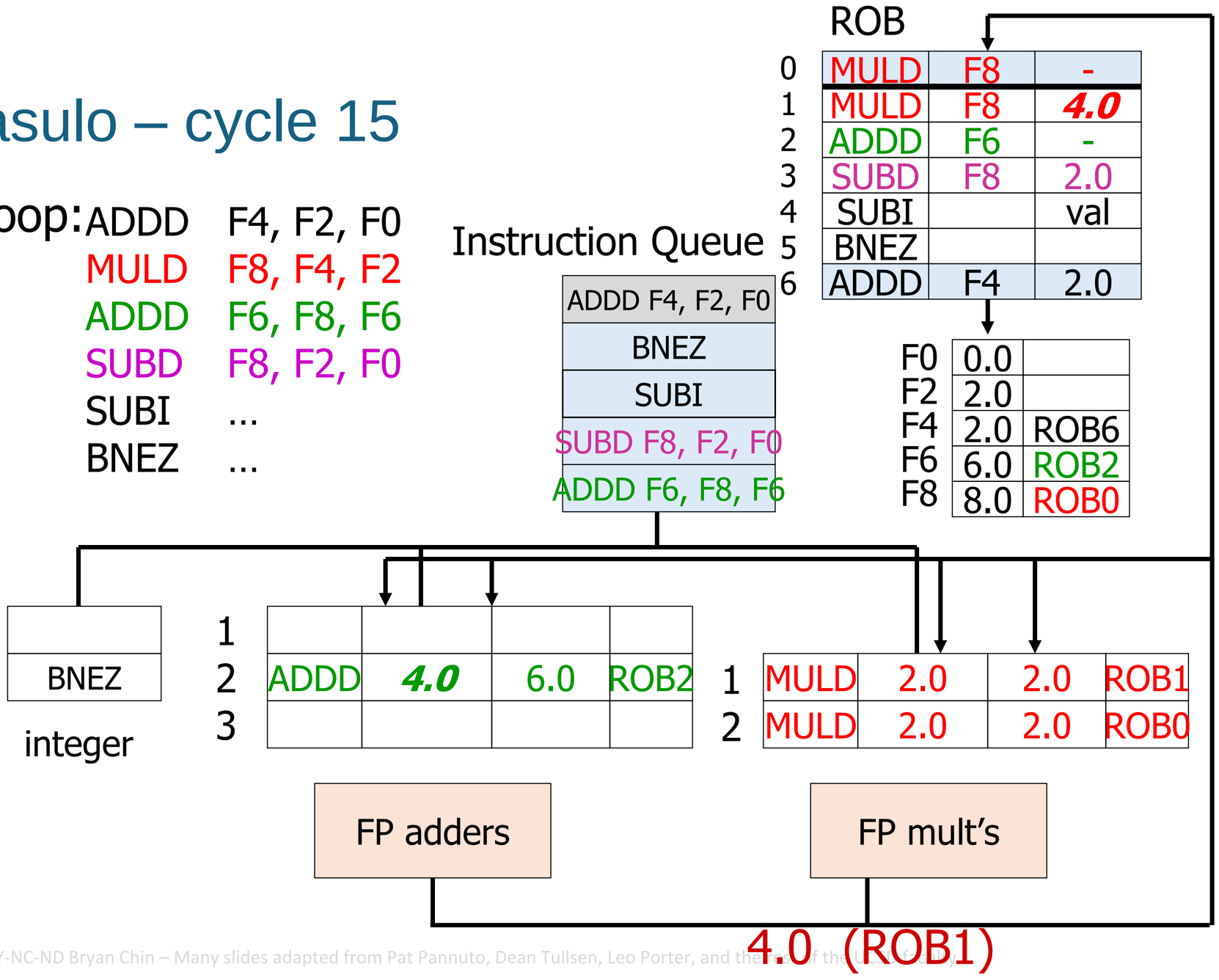
Tomasulo – cycle 11

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



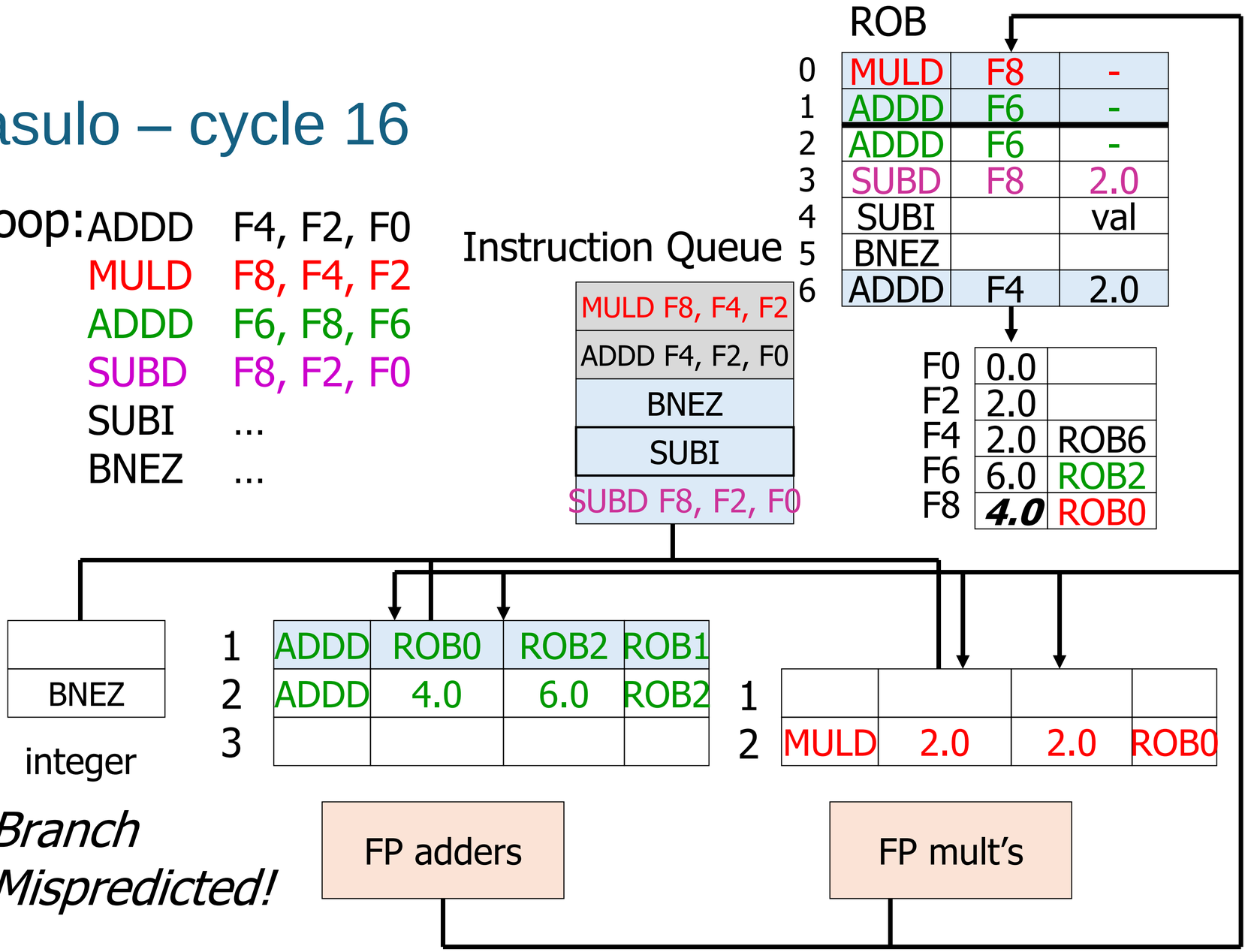
Tomasulo – cycle 15

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



Tomasulo – cycle 16

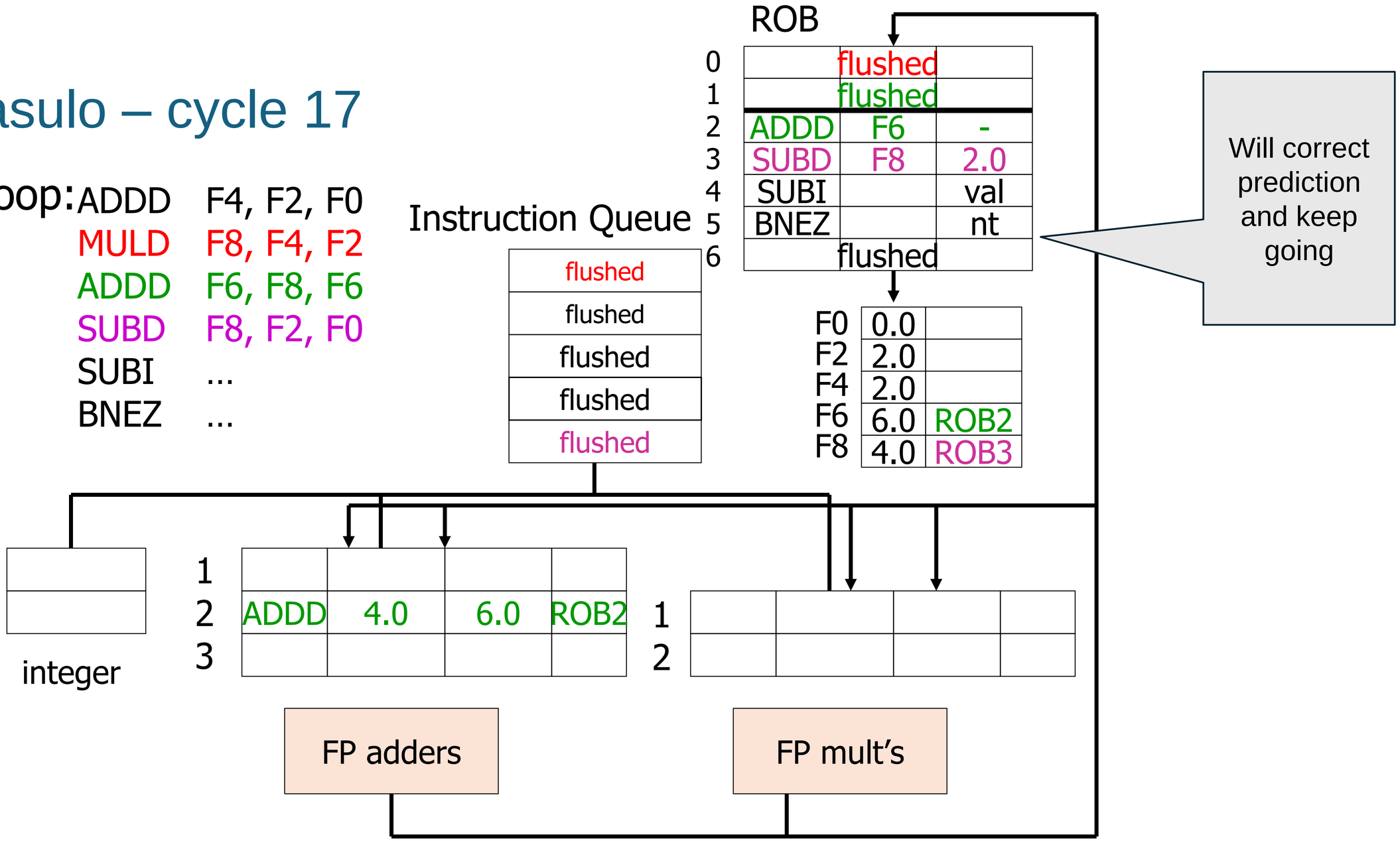
```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



Branch Mispredicted!

Tomasulo – cycle 17

```
Loop: ADDD  F4, F2, F0
      MULD  F8, F4, F2
      ADDD  F6, F8, F6
      SUBD  F8, F2, F0
      SUBI  ...
      BNEZ  ...
```



Note

- For this to work, we have to assume some checkpointing of the register result status table!

F0	0.0	
F2	2.0	
F4	2.0	
F6	6.0	ROB2
F8	4.0	ROB3

Speculative Execution

- The re-order buffer and in-order commit allow us to flush the speculative instructions from the machine when a misprediction is discovered.
- ROB is another possible source of operands
- ROB can provide *precise exceptions* in an out-of-order machine
- ROB allows us to *ignore exceptions* on speculative code.

Dynamic Scheduling Key Points

- Dynamic scheduling is code motion in HW.
- Dynamic scheduling can do things SW scheduling (static scheduling) cannot.
- Register renaming eliminates WAW, WAR dependencies.
- To get cross-iteration parallelism, we need to eliminate WAW, WAR dependencies.