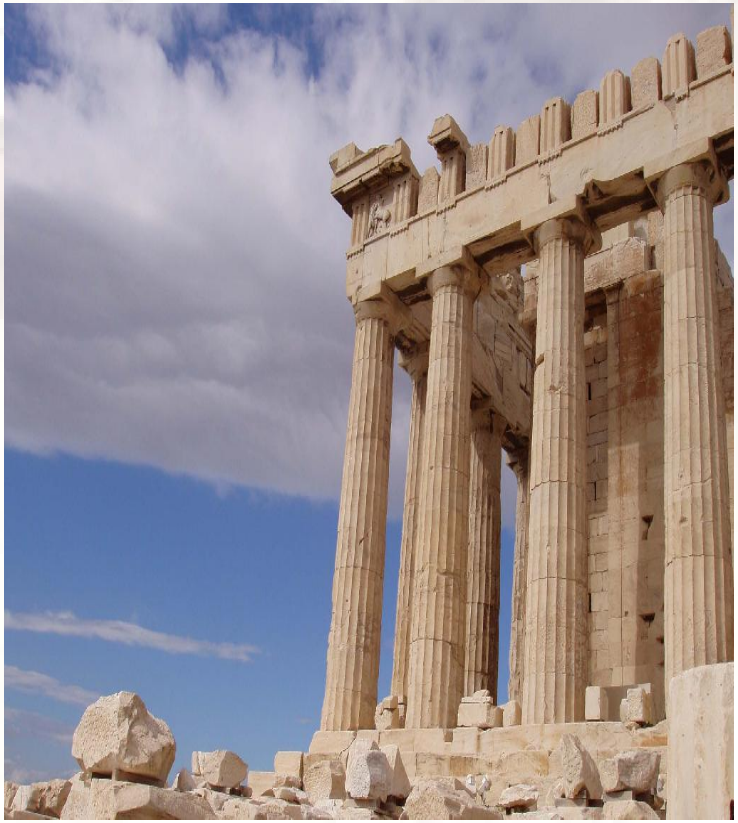
Ch3-3

ILP: Branch Prediction & Hardware-based speculation



浙江大學

ZHEJIANG UNIVERSITY

Review

- ❑ Structural Hazard
  - Separate/replicate function unit, fully-pipelined function units
- ❑ Data Hazard
  - RAW: double-bump, forwarding path, simple compiler scheduling
  - WAR, WAW: implicit register renaming
- ❑ Control Hazard
  - Resolve branch asap, Flushing, Predict-not-taken, Predict-taken, Delayed-Branch

Control Hazard (example P_{A-25})

- **Flushing:** stall until the branch is resolved.
  - Stall X cycles if delay slots=x
- **Predict-not-taken:**
  - Hardware takes all branches as not-taken, compiler need to put frequent case in the not-taken branch
  - Stall 0 cycles if not-taken, stall X cycles if taken
- **Predict-taken:**
  - Hardware takes all branches as taken, can go forward only when get the branch target . Compiler need to put frequent case in the taken branch
  - Stall y cycles waiting for branch target; stall X cycles if not taken.
- **Delayed Branch:**

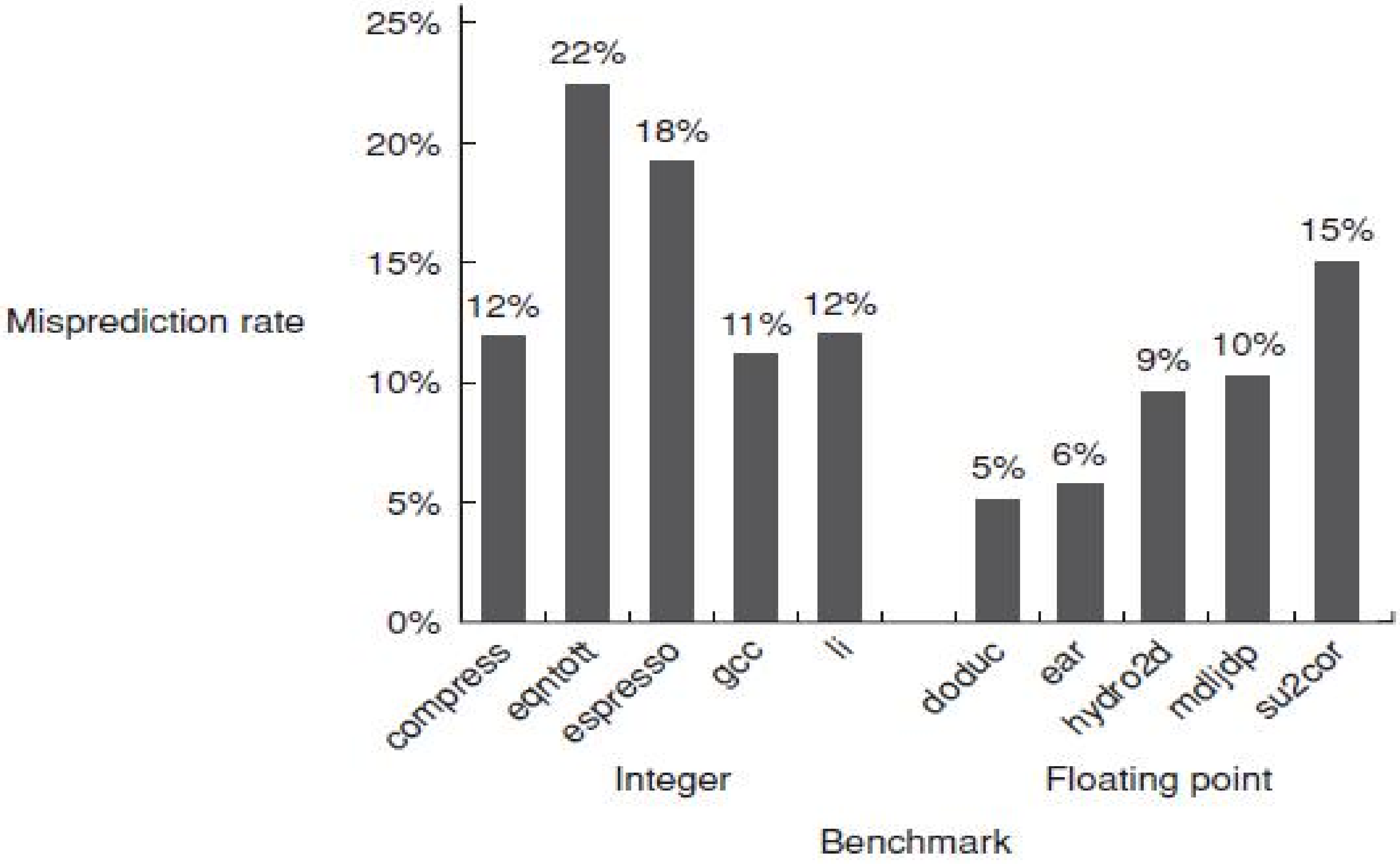
Static branch prediction

- □Stall for control hazards

  = f (branch frequency, prediction accuracy, misprediction penalty)

- □Predict according to statistics: (predict taken)
  - ➢Mispredition: 9% ~ 59%, average 34%

Predict based on profile information



Average misprediction rate 9% for FP programs,

Average misprediction rate 15% for integer programs

Dynamic Hardware Prediction --reducing branch costs

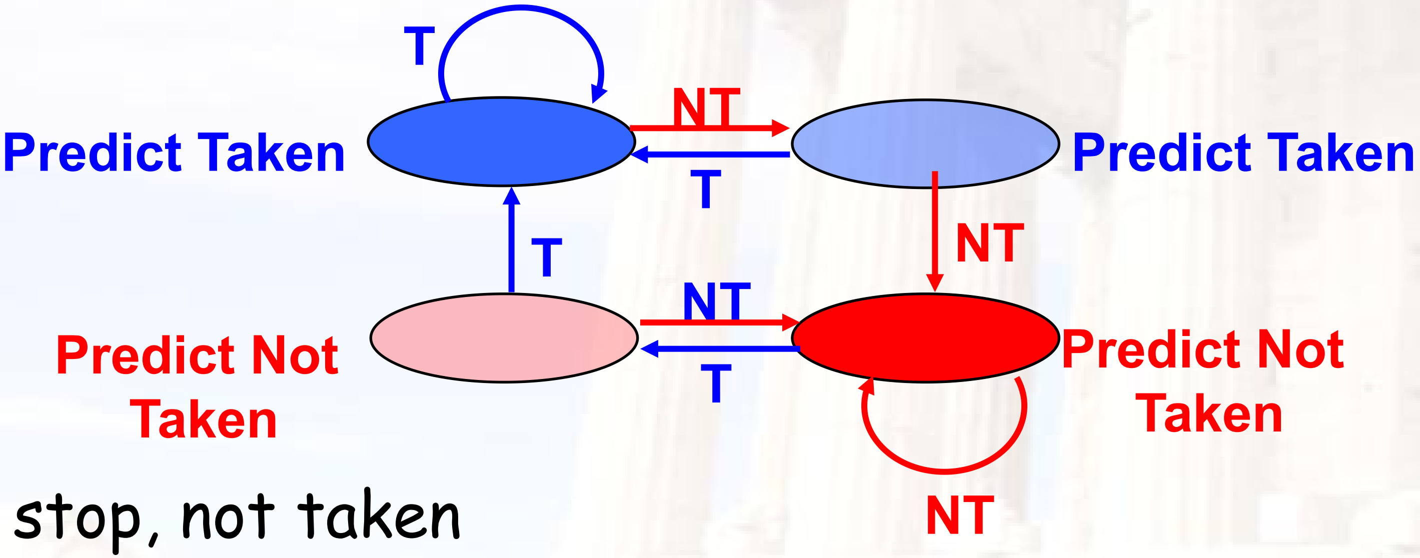
1. 1-bit Branch-Prediction Buffer
2. 2-bit Branch-Prediction Buffer
3. Correlating Branch Prediction Buffer
4. Tournament Branch Predictor
5. Branch Target Buffer
6. Integrated Instruction Fetch Units
7. Return Address Predictors

1-bit Branch-Prediction Buffer

- Performance = *f*(accuracy, cost of misprediction)
- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check (saves HW, but may not be right branch)
- Problem: in a loop, 1-bit BHT will cause 2 mispredictions (avg 2/n iterations before exit):
  - Only (n-2)/n accuracy even if loop (n-1)/n of the time
  - 0111……1110111… …1110111……1110111…

2-bit Branch-Prediction Buffer

- ❑ Solution: 2-bit scheme where change prediction only if get misprediction *twice*:
  - ➢ (Jim Smith, 1981, A study of branch prediction strategies)



Predict Taken

Predict Taken

Predict Not Taken

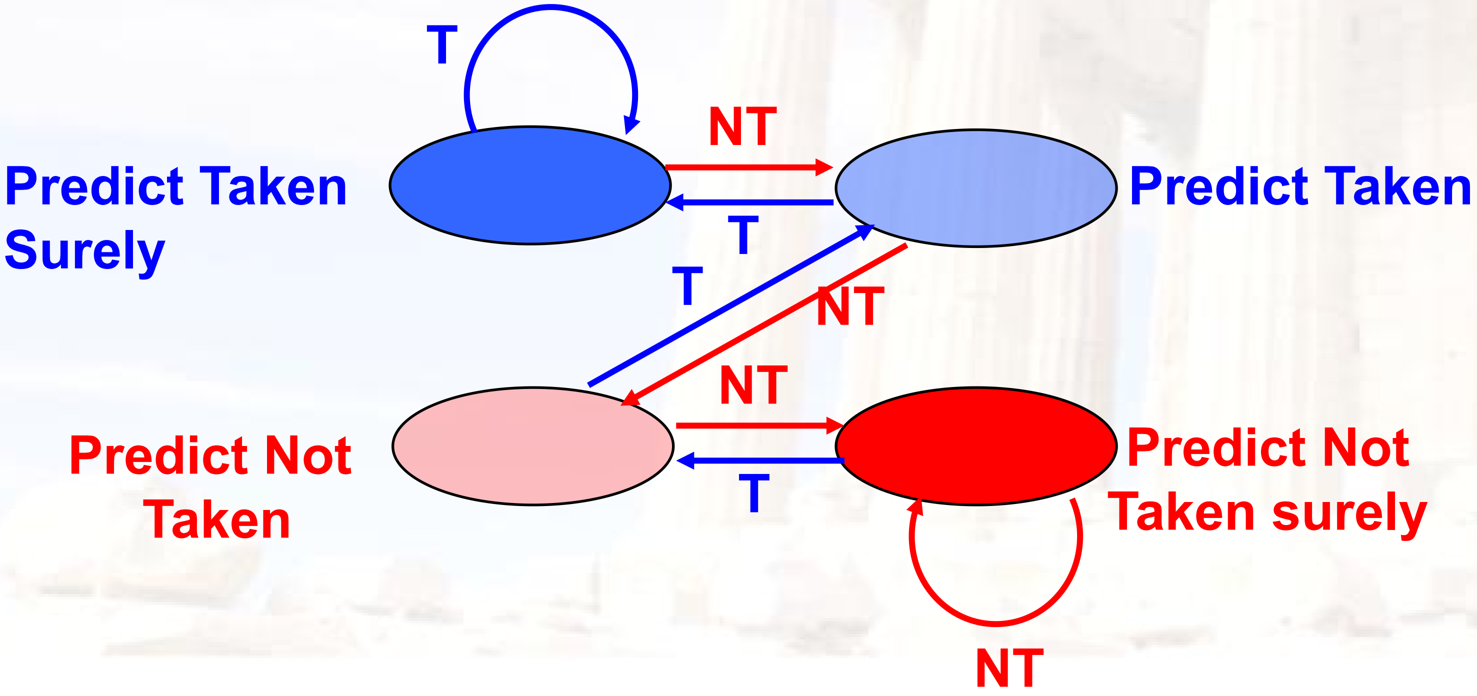
Predict Not Taken

- ❑ Red: stop, not taken
- ❑ Blue: go, taken
- ❑ Adds *hysteresis* to decision making process

➢ 01111111……111110111111…1110111111

An alternative 2-bit predictor

- It can be implemented using a 2-bit counter



Generalize the 2-bit predictor to n-bit predictor

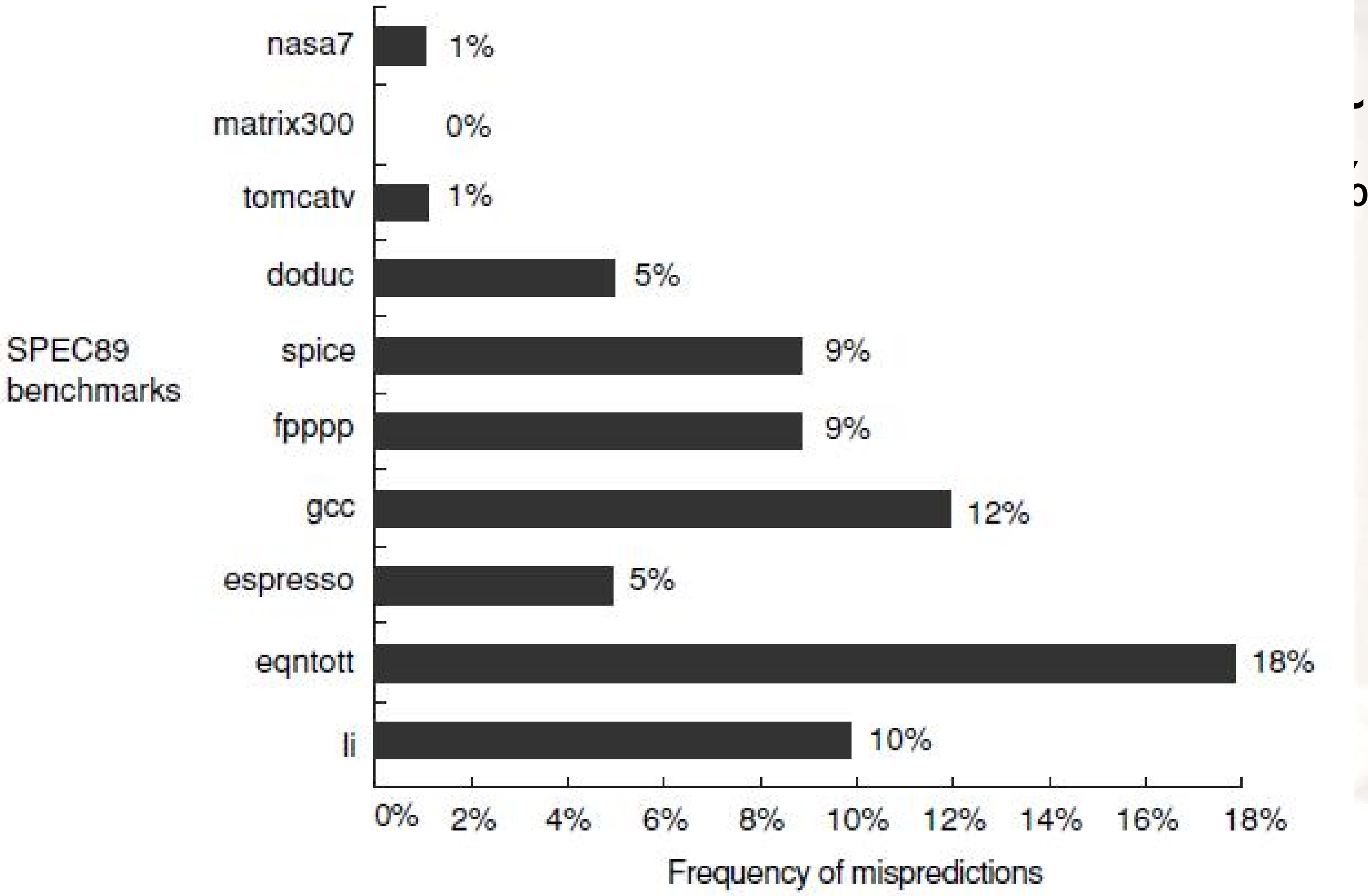
- N-bit counter ($0<x<2^{n}-1$)
- Predict taken $X >= 2^{n-1}$
- Predict untaken $X < 2^{n-1}$
- Counter + 1 when a taken branch
- Counter – 1 when a not taken branch
- Most system rely on 2-bit predictor rather than n-bit predictor.

Accuracy of the 2-bit predictor

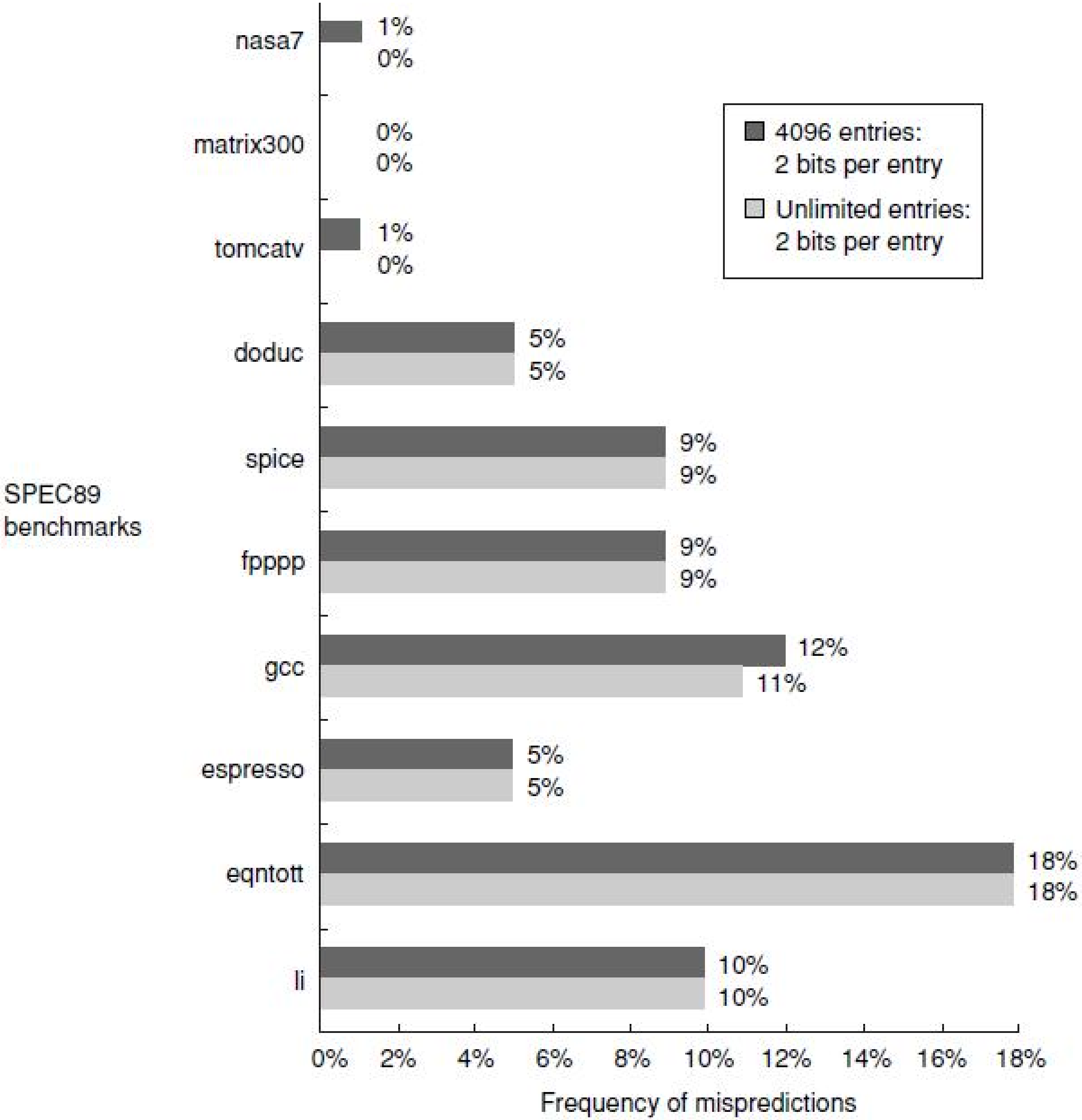
- 4096 entries, 2-bit predictor

-82%

~18%



How about unlimited entries ?



How to improve accuracy ?

❑Analysize the branch behavior

```
IF ( aa == 2)
      aa = 0;
IF ( bb == 2)
      bb = 0;
IF (aa == bb )
   {
   ……
}
```

```
     SUBI   R3, R1, #2
     BNEZ  R3, L1          ; br.b1 (aa!=2)
     ADD    R1, R0, R0  ; aa==0
L1: SUBI   F3, R2, #2
     BNEZ  R3, L2          ; br.b2 (bb!=2)
     ADD    R2, R0, R0 ; bb==0
L2:  SUB    R3, R1, R2;  R3=aa-bb
     BEQZ R3, L3        ; br.b3
```

Conclusion from the example

- ❑B3 branch depends on b1 and b2. B3 is taken only when both b1 and b2 is untaken.
- ❑It's impossible to predict B3 correctly only depending on its previous behavior.
- ❑A branch's behavior not only rely on it's previous behavior but also on recent behavior of other branches.

Let’s check an example

```
If (d==0)
    d=1;
if (d==1){
   ……

}
```

❑ **Assume Reg[R1] = d**

```
    BNEZ       R1, L1         ;  br b1, (d!=0)
    DADDIU   R1, R0, #1     ; d==0, so  d=1
L1: DADDIU     R3, R1, #-1    ;
    BNEZ       R3, L2         ;  br b2, (d!=1)
      ……
L2:
```

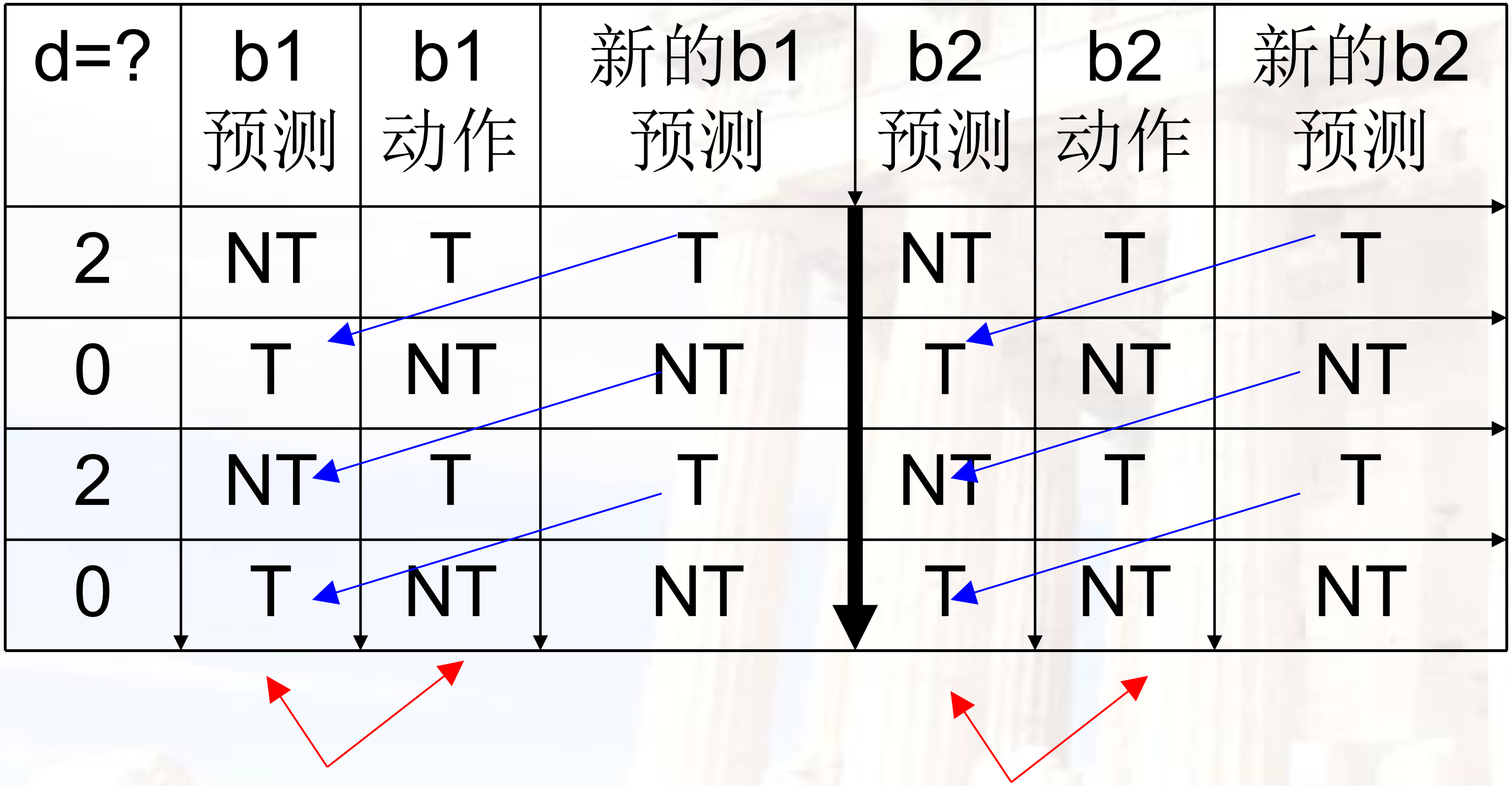
Feature of the code:

d的初值	d==0?	B1	在b2以前的d值	d==1?	b2
0	Yes	Not taken	1	Yes	Not taken
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	no	Taken

Conclusion: b2 will be **not taken** if **b1 is not taken**

So, we can get accurate prediction if using correlate predictor.

Always mispredict if using one-bit predictor



d=?	b1 预测	b1 动作	新的b1 预测	b2 预测	b2 动作	新的b2 预测
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

预测与实际动作总是相反

预测与实际动作总是相反

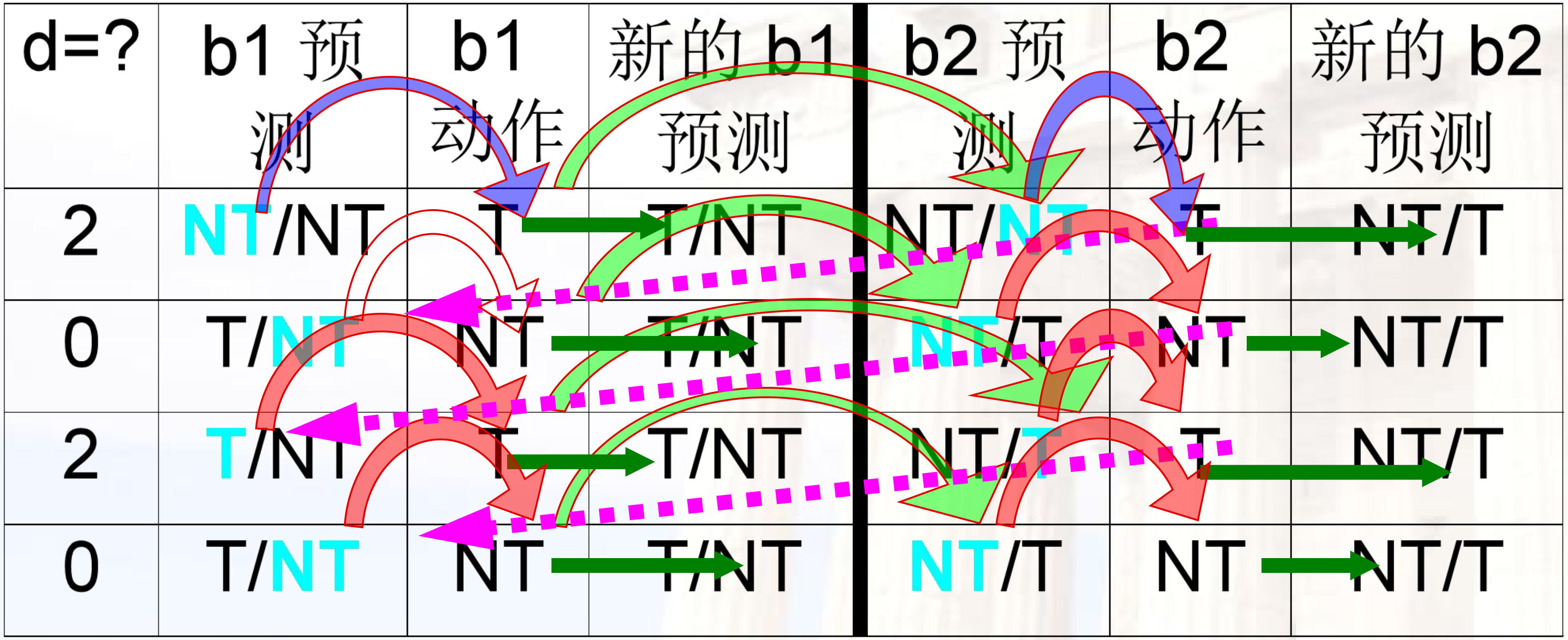
Correlating predictor

- ❑Correlating predictor is composed of two parts (two bits):
  - 1st bit: prediction for previous br. is NT,
  - 2nd bit: prediction for previous br. Is T
- ❑Four combinations

预测组合	上一次Br为NT,预测本次位(看第一位)	上一次Br为T,预测本次为(看第二位)
NT/NT	NT	NT
NT/ T	NT	T
T /NT	T	NT
T / T	T	T

Note, previous Br might be a different br.

Prediction with correlating predictor (1,1) with initial value =NT/NT.



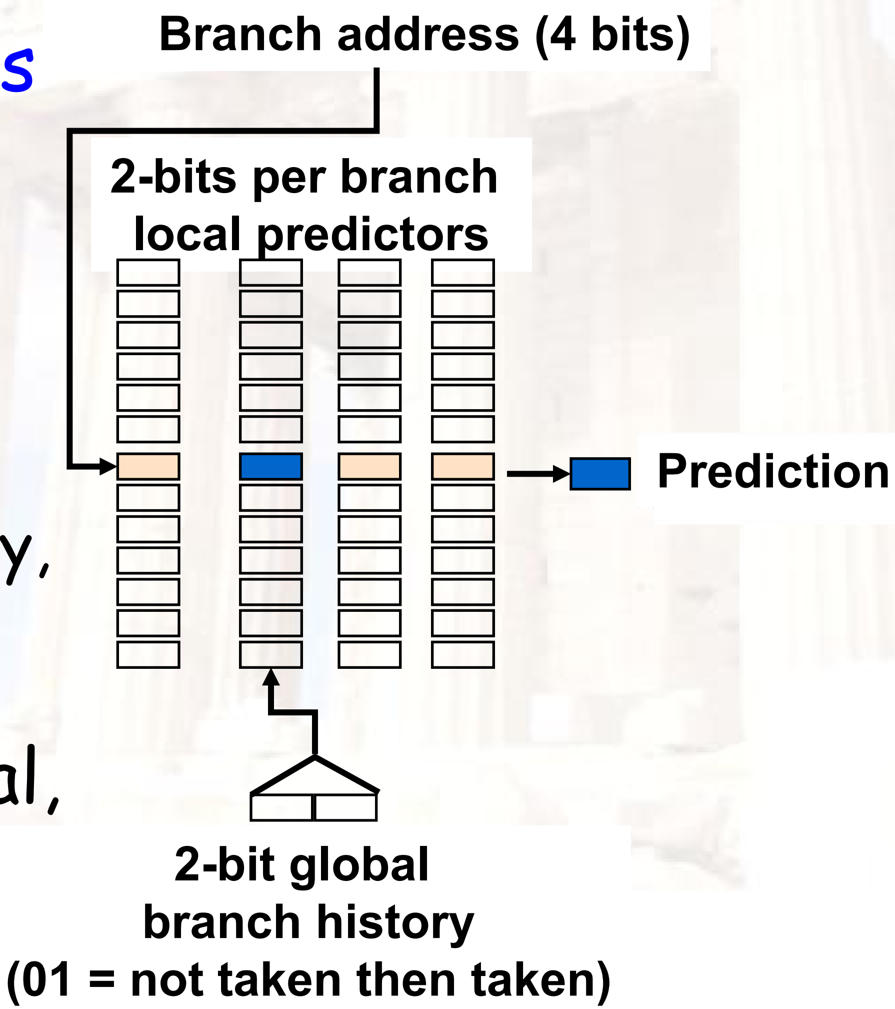
d=?	b1 预测	b1 动作	新的b1 预测	b2 预测	b2 动作	新的 b2 预测
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

粗体字表示本次预测

🟦	预测出错	🟩	**b1动作与b2预测值选用关系**
🟥	预测正确	🟪	**b2动作与b1预测值选用关系**
🟩	新预测值		

Correlating Branches prediction buffer

- Idea: taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)
  - Then behavior of recent branches selects between, say, 4 predictions of next branch, updating just that prediction
- (2,2) predictor: 2-bit global, 2-bit local

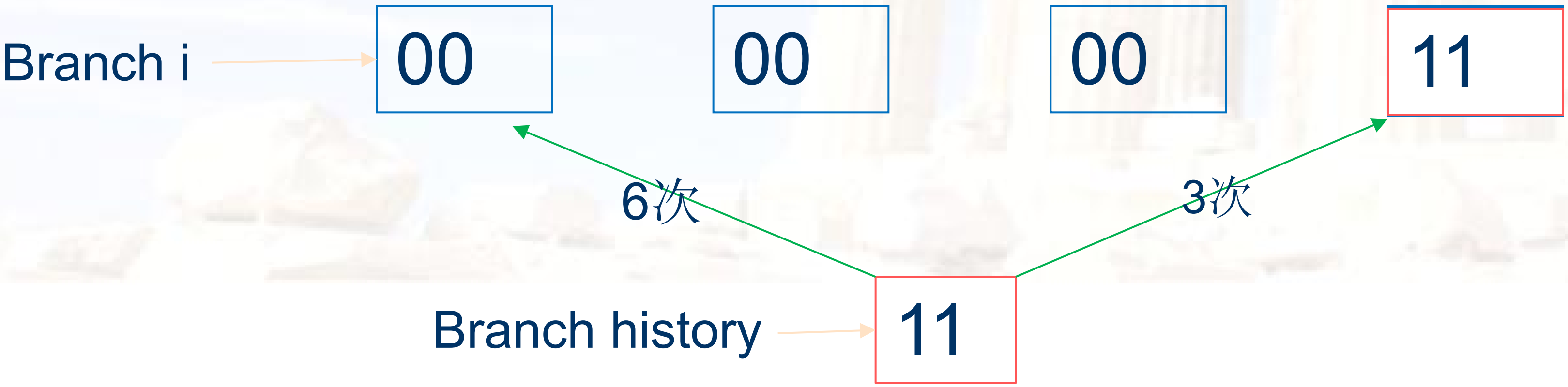

Branch address (4 bits)
2-bits per branch local predictors
Prediction
2-bit global branch history
(01 = not taken then taken)

4.Correlating Branches prediction buffer

Assume branch history with branch i in blue box.

0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1

Predict: v v x V V X V V V



Branch i

00

00

00

11

6次

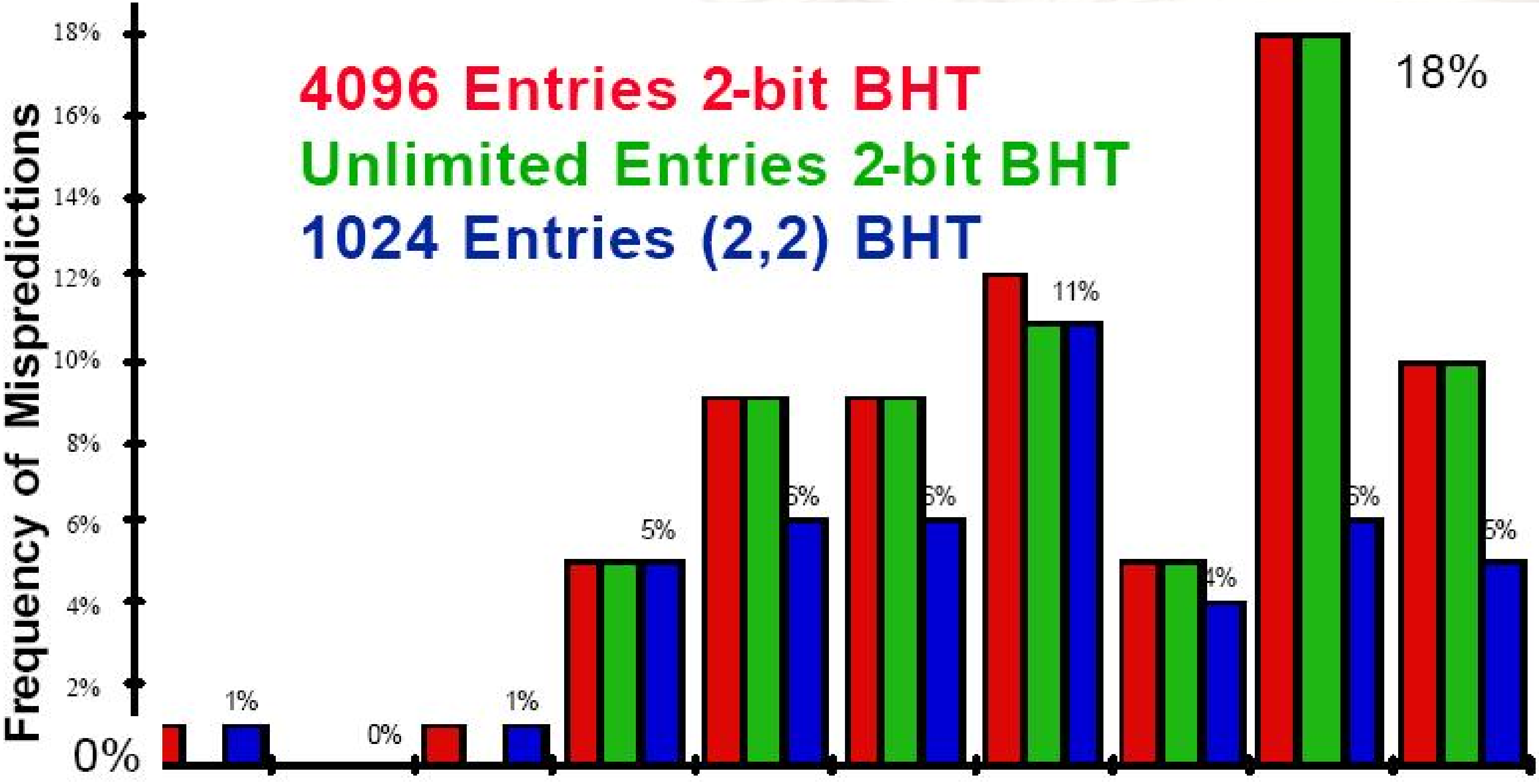
3次

Branch history

11

- (m, n) predictor
- m: previous m branch instruction
- 2^m

Comparision of 2-bit predictors



Accuracy analysis

❑Mispredict because either:

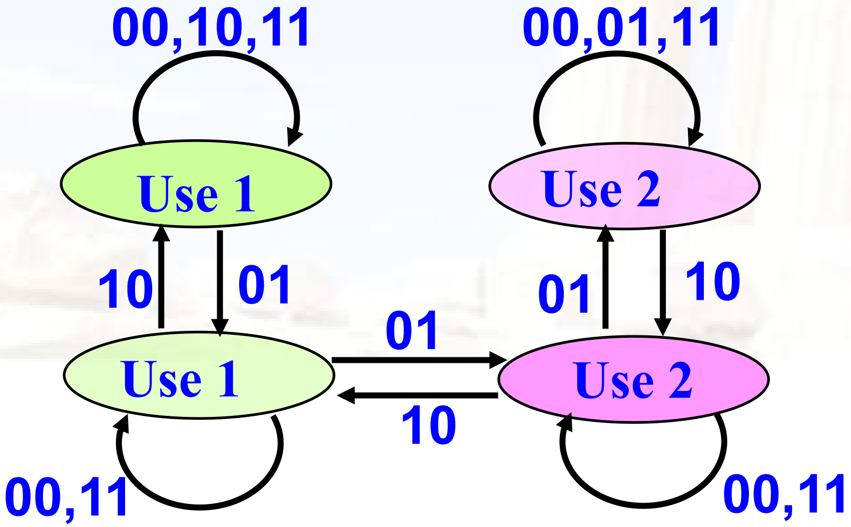
- ➢Wrong guess for the branch
- ➢Got branch history of wrong branch when index the table.

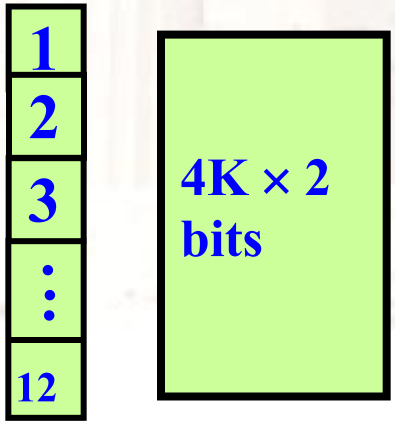
4、 Tournament Predictors

- Motivation for correlating branch predictors is 2-bit predictor failed on important branches; by adding global information, performance improved
- Tournament predictors: use 2 predictors, 1 based on global information and 1 based on local information, and combine with a selector
- Hopes to select right predictor for right branch

Tournament Predictor in Alpha 21264

- ❑ 4K 2-bit counters to choose from among a global predictor and a local predictor
- ❑ Global predictor also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor
  - ➢ 12-bit pattern: ith bit 0 => ith prior branch not taken; ith bit 1 => ith prior branch taken;

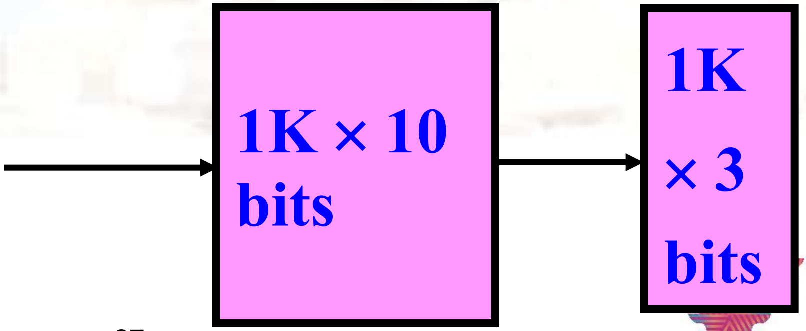




T-Predictor in Alpha 21264 (cont.)

- Local predictor consists of a 2-level predictor:
  - Top level a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted.
  - Next level Selected entry from the local history table is used to index a table of 1K entries consisting a 3-bit saturating counters, which provide the local prediction

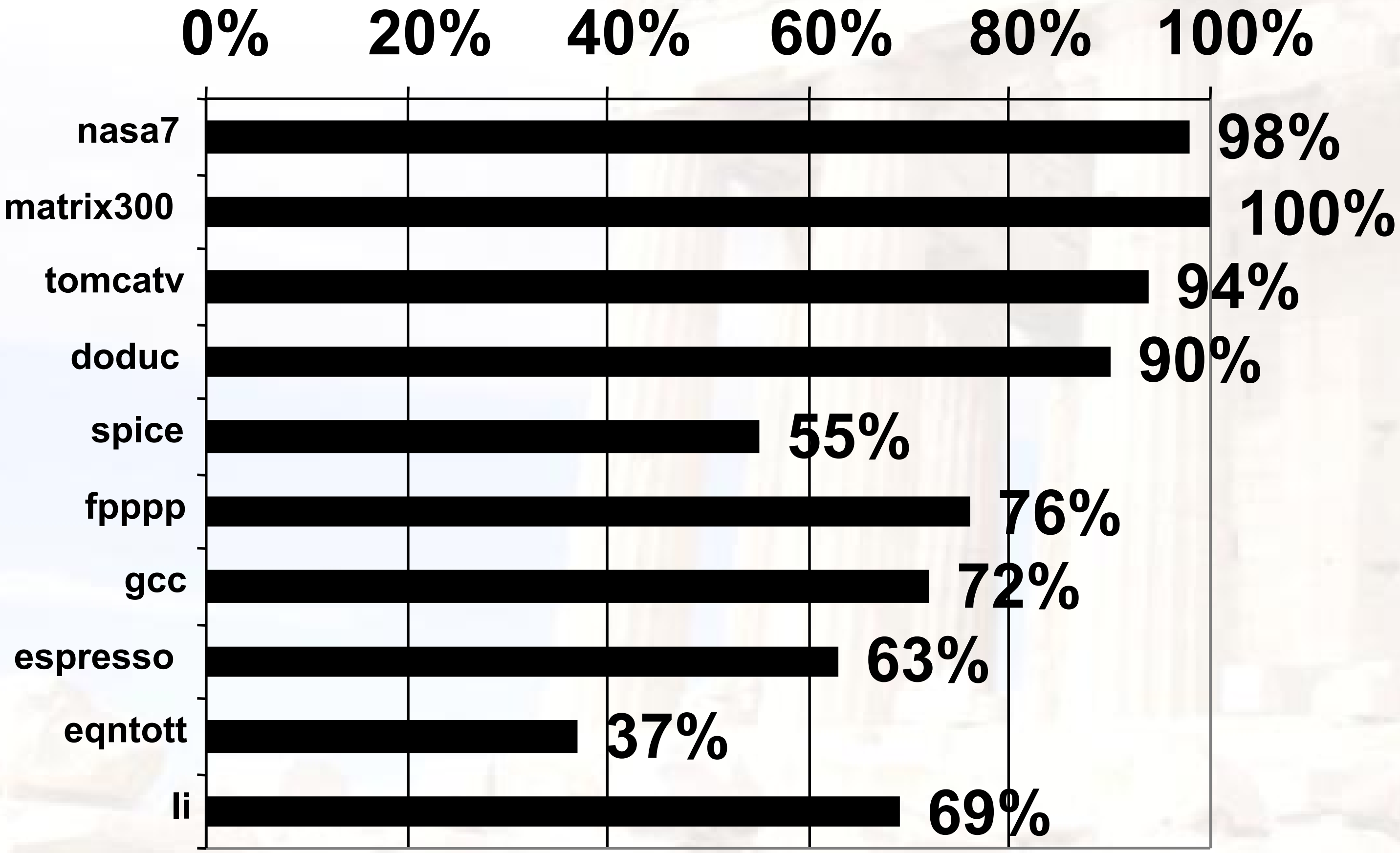
- Total size: 4K*2 + 4K*2 + 1K*10 + 1K*3 = 29K bits! (~180,000 transistors)



1K × 10 bits

1K × 3 bits

% of predictions from local predictor in Tournament Prediction Scheme



Accuracy of Branch Prediction

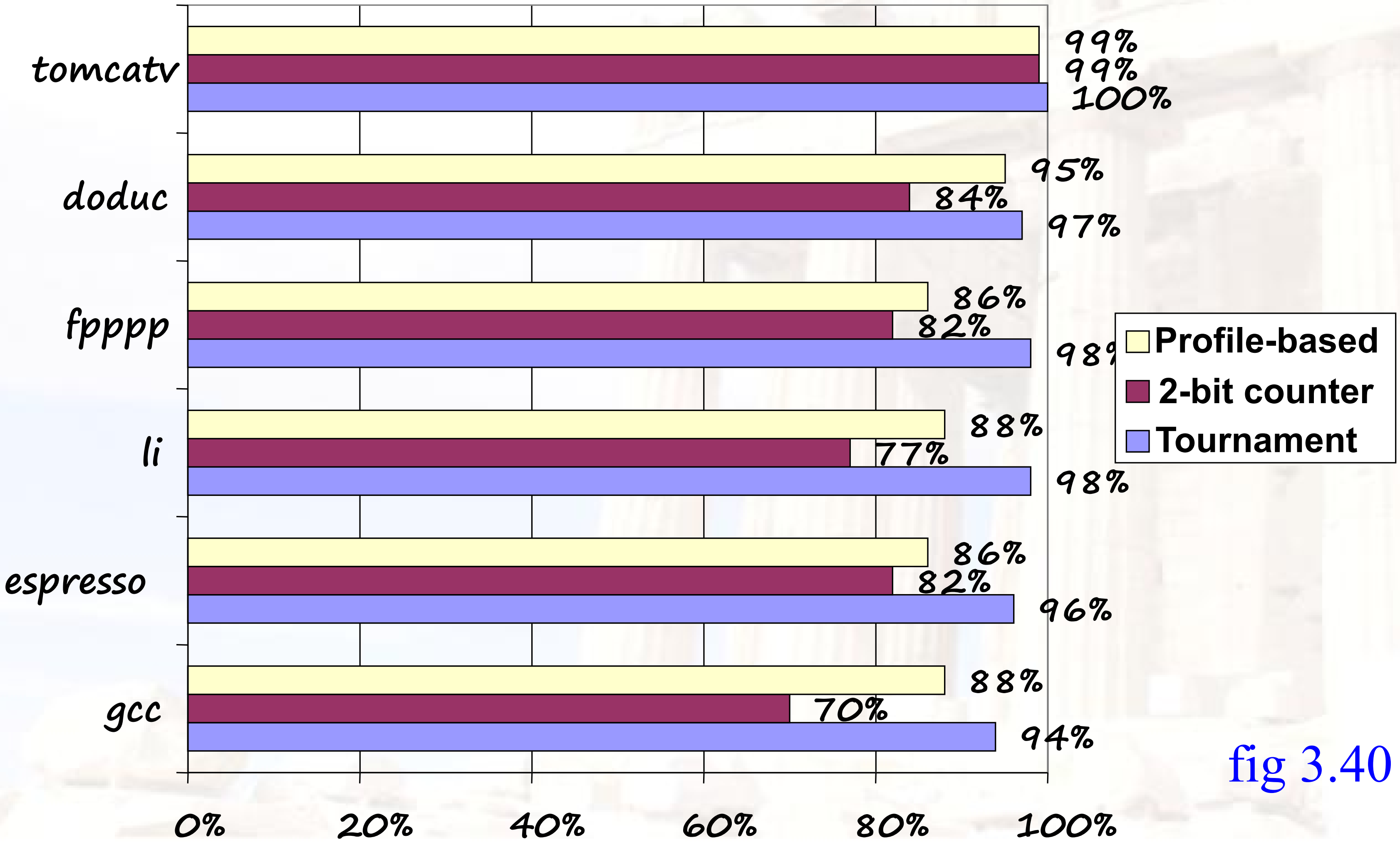
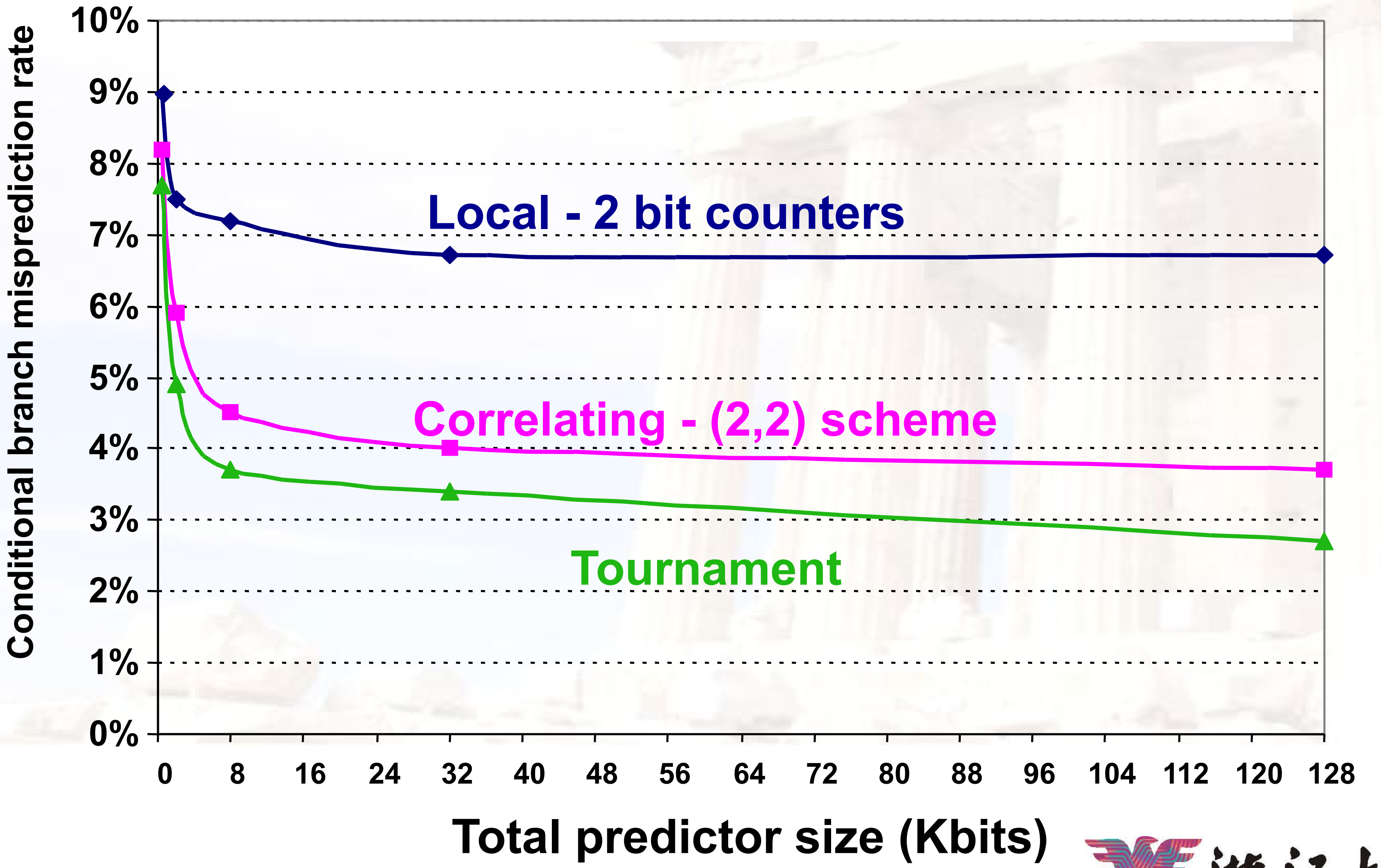


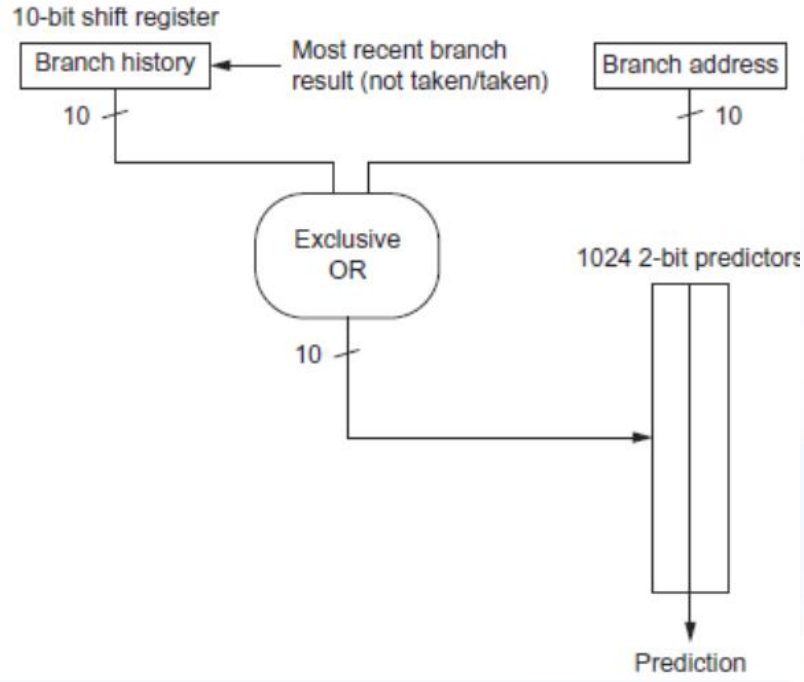
fig 3.40

- Profile: branch profile from last execution (static in that is encoded in instruction, but profile)

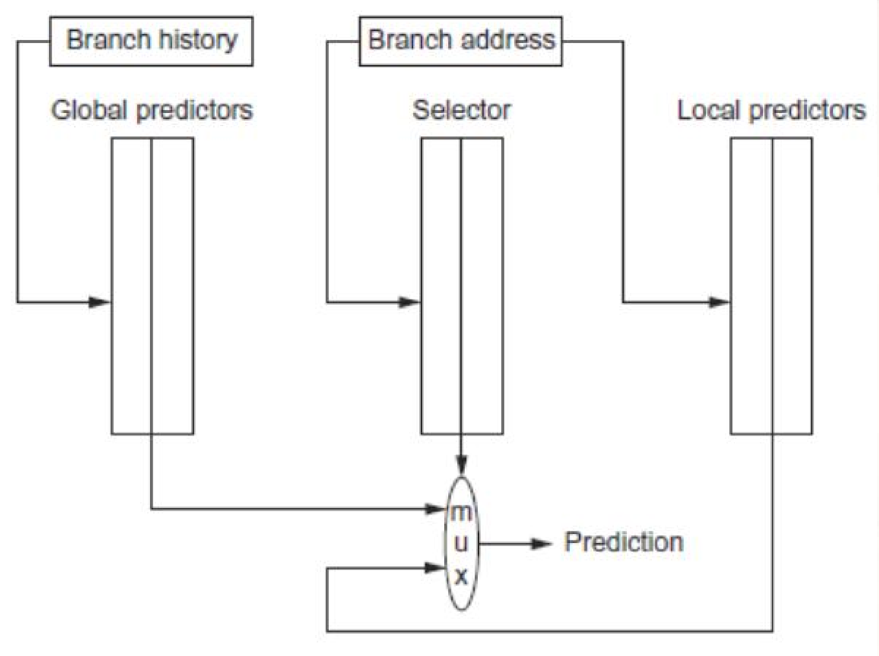
Accuracy v. Size (SPEC89)



Other Branch Prediction

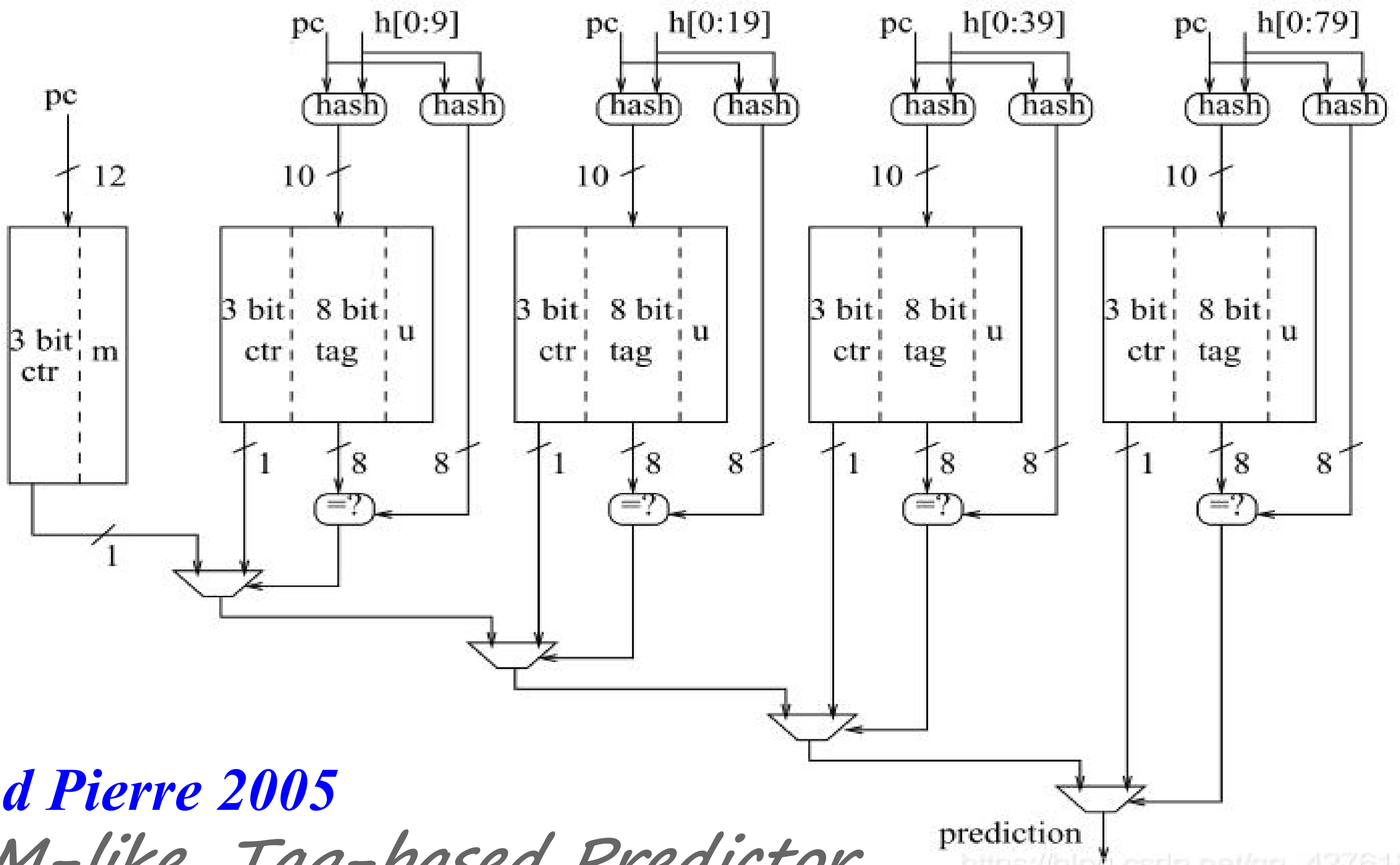


gshare



tournament

PPM（Prediction by Partial Matching）



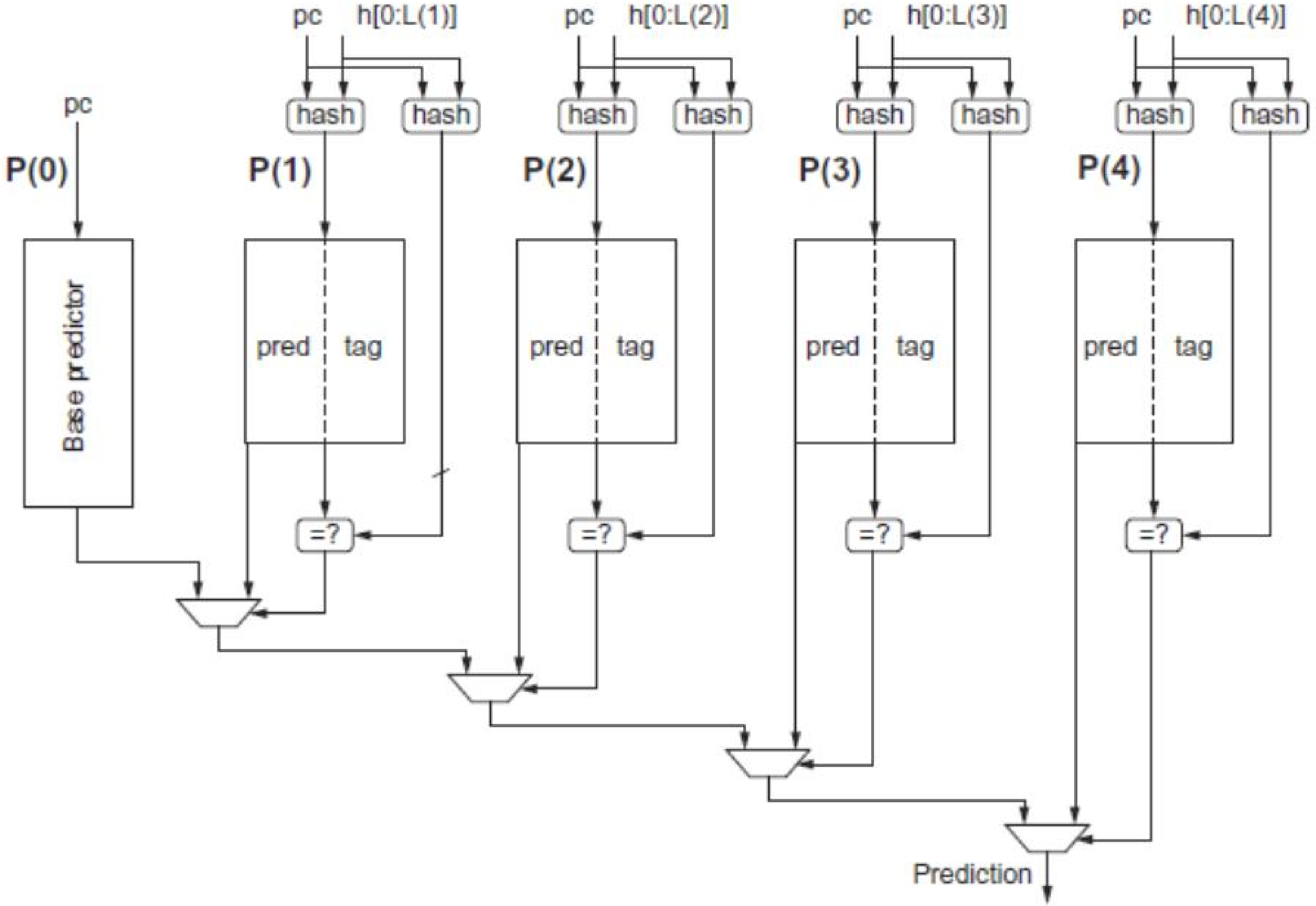
Michaud Pierre 2005

- *A PPM-like, Tag-based Predictor.*
- *J. Instr. Level Parallelism 7 (2005)*
- 第一届CBP取得了第五名的成绩

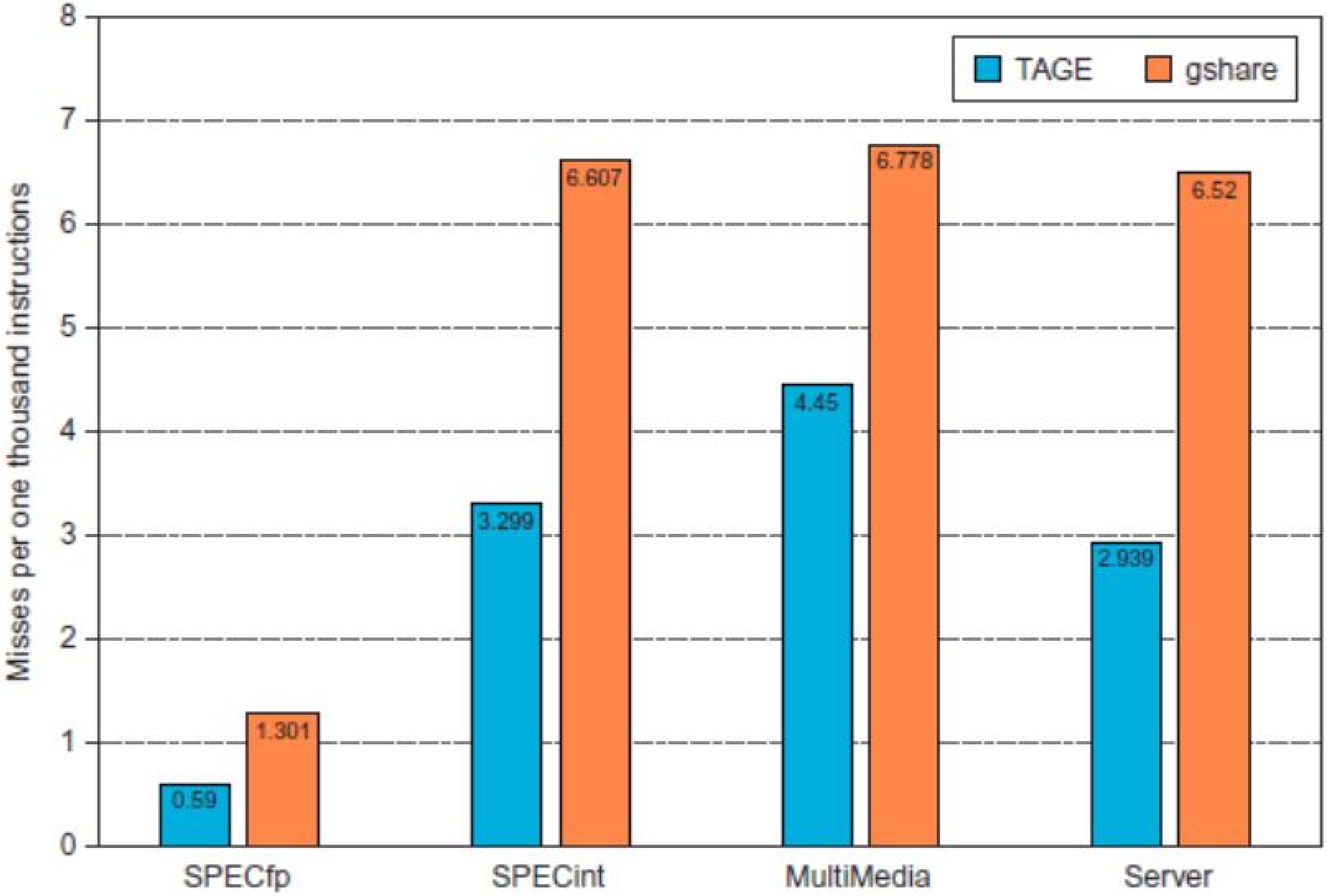
Tagged Hybrid Predictors

- Need to have predictor for each branch and history
  - Problem: this implies huge tables
  - Solution:
    - Use hash tables, whose hash value is based on branch address and branch history
    - Longer histories may lead to increased chance of hash collision, so use multiple tables with increasingly shorter histories

TAGE(TAgged GEometric history length branch prediction:

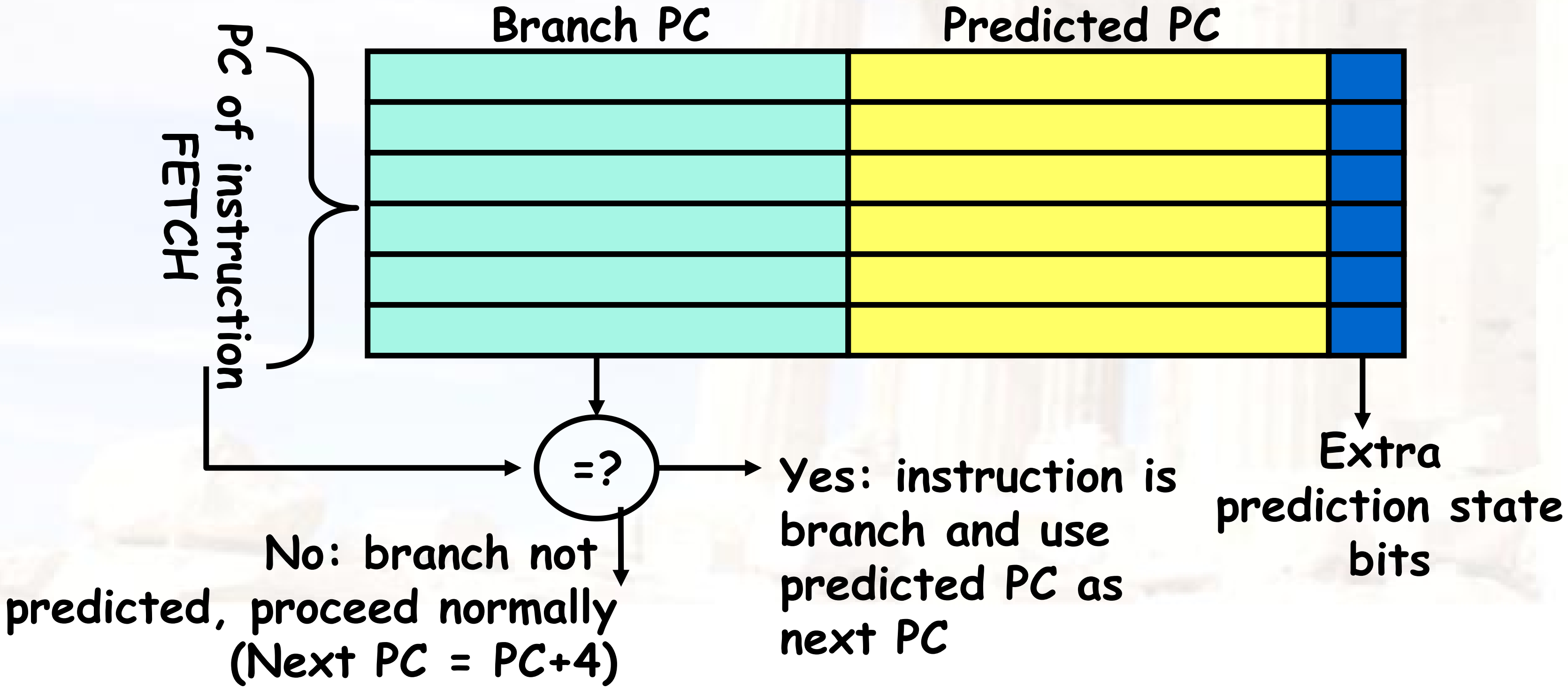


Tagged Hybrid Predictors



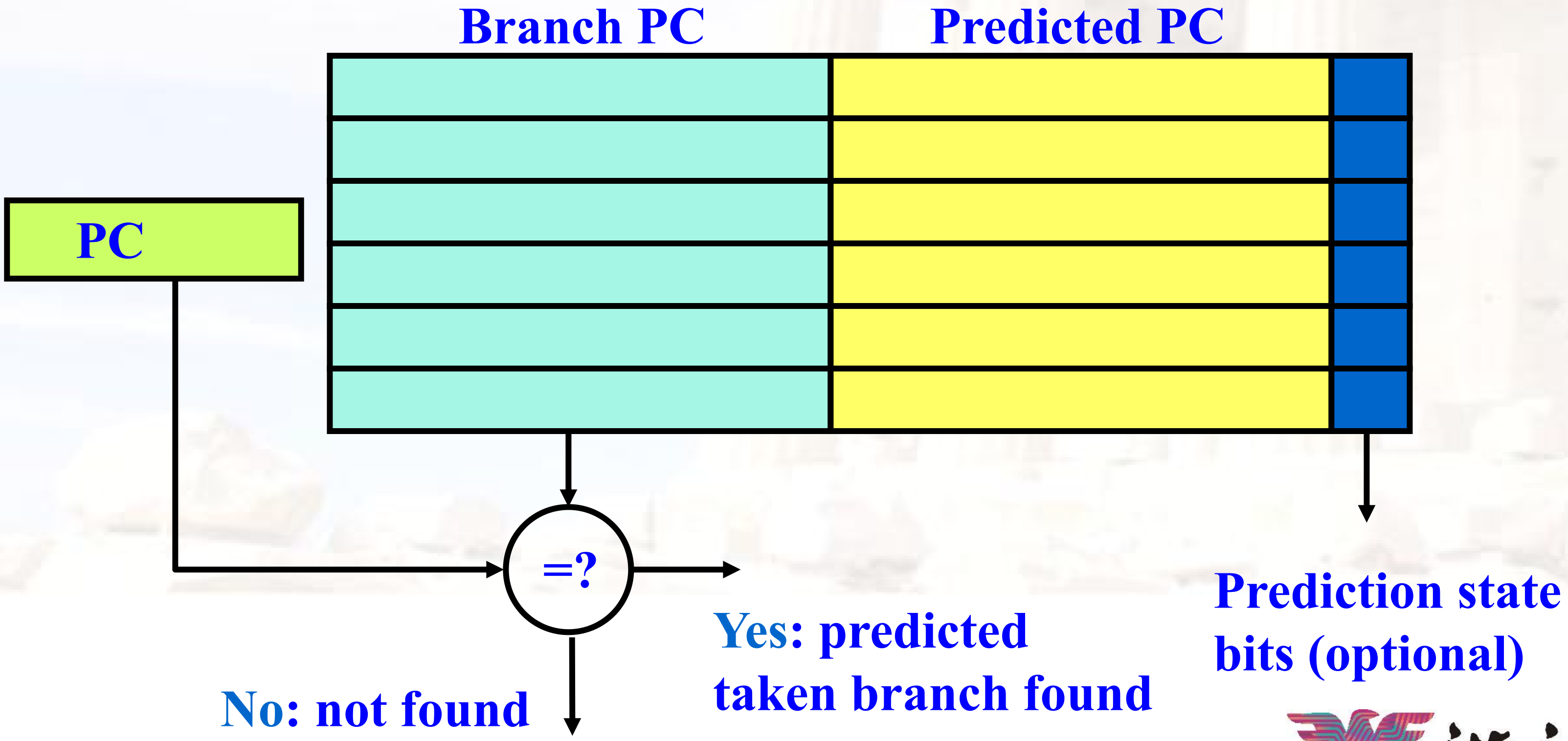
5. Branch Target Buffer

- Branch Target Buffer (Branch Target Cache):
  - Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address

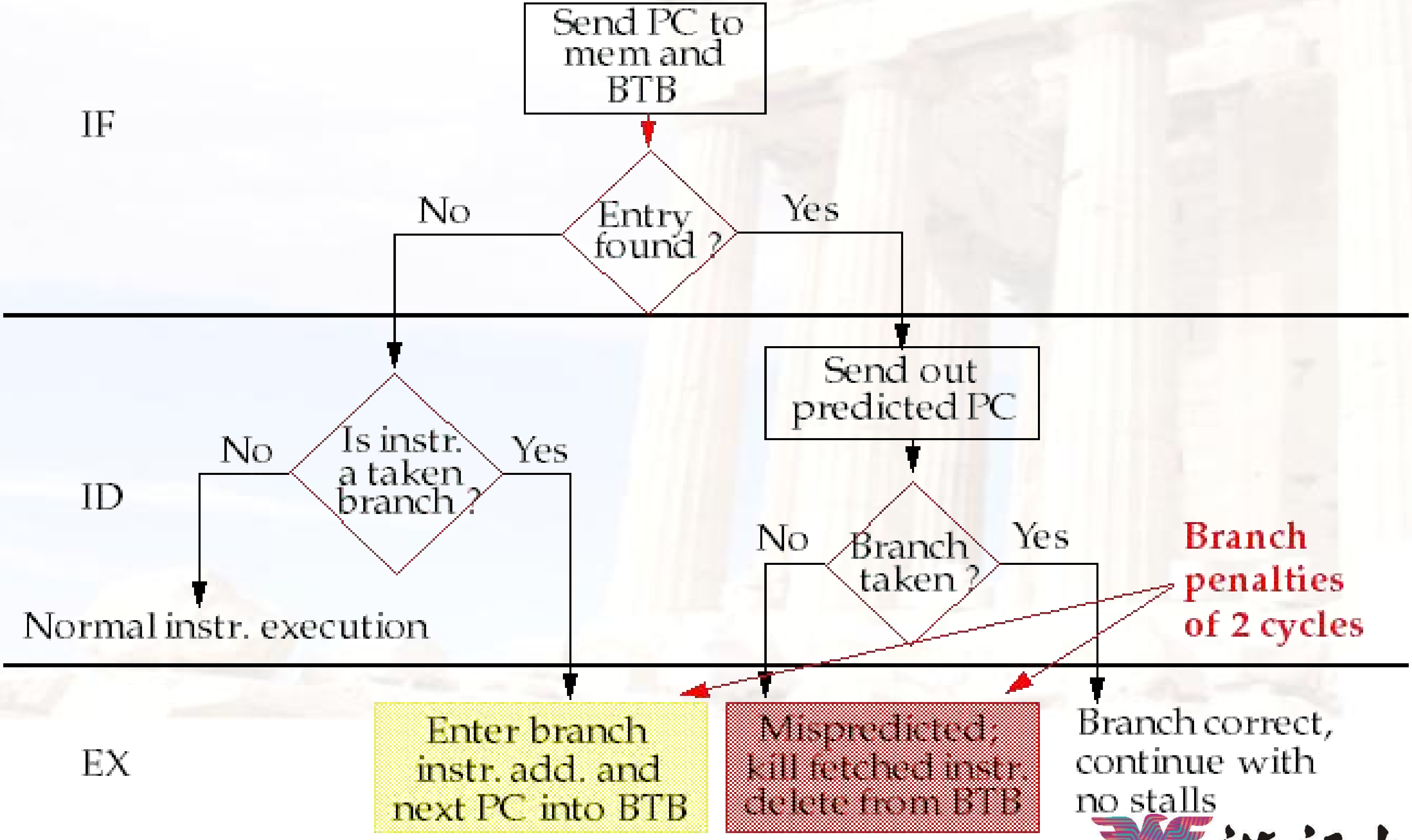

Branch PC
Predicted PC
PC of instruction FETCH
=?
Yes: instruction is branch and use predicted PC as next PC
No: branch not predicted, proceed normally (Next PC = PC+4)
Extra prediction state bits

Branch Target “Cache”

- Branch Target cache - Only predicted taken branches
- “Cache” - Content Addressable Memory (CAM) or Associative Memory (see figure)
- Use a big Branch History Table & a small Branch Target Cache


Branch PC
Predicted PC
PC
=?
No: not found
Yes: predicted taken branch found
Prediction state bits (optional)

Steps in handling an instruction with a Branch-Target Buffer



IF
Send PC to mem and BTB
Entry found ?
No
Yes
ID
Is instr. a taken branch ?
No
Yes
Send out predicted PC
Branch taken ?
No
Yes
Normal instr. execution
EX
Enter branch instr. add. and next PC into BTB
Mispredicted, kill fetched instr. delete from BTB
Branch correct, continue with no stalls

Branch penalties of 2 cycles

Variations of Branch Target Buffer

- □Instead of storing just the branch address, the BTB can store the actual instruction as well.
- □Storing both the branch address and the actual instruction in the buffer
- □Very big branch Target buffer to cache addresses or instructions from multiple paths, ex, the predicted and unpredicted direction

6. Integrated Instruction Fetch Units

- ❑Increasing the number of instructions executed per clock, instruction fetch will become an significant bottleneck.
- ❑Integrated instruction fetch unit:
  - ➢Integrated branch prediction
  - ➢Instruction prefetch

    (see P438 hardward prefetch &

    P483 ex. Instruction prefetcher)
  - ➢Instruction memory access and buffering

7、Return Address Predictors

- Technique for predicting indirect jumps, whoes destination address varies at run time.
- Register Indirect branch hard to predict address
- SPEC89: procedure return account for 85% of the indirect jumps.
- Branch-target buffer doesn't work well for procedure return

Return Addresses Buffer

- Since stack discipline for procedures, save return address in small buffer that acts like a stack:
  - Caches the most recent return addresses.
  - Push return address into stack at a call;
  - pop an address off at a return

- SPEC benchmark shows a stack with 8-16 entries works quite well. (fig-3.22)

Dynamic Branch Prediction Summary

- Prediction becoming important part of scalar execution
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch.
  - Either different branches
  - Or different executions of same branches
- Tournament Predictor: more resources to competitive solutions and pick between them
- Tagged Hybrid Predictors: have predictor for each branch and history
- Branch Target Buffer: include branch address & prediction
- Predicated Execution can reduce number of branches, number of mispredicted branches
- Return address stack for prediction of indirect jump

Chapter3 ILP

Hardware Based Speculation

Tomasulo Drawbacks

- Complexity
  - delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620 in CA:AQA 2/e, but not in silicon!
- Many associative stores (CDB) at high speed, Performance limited by Common Data Bus
- Non-precise interrupts!
  - We will address this later
- Partially overlapped basic blocks
  - integer units get ahead, beyond FP operation
  - the successor basic block can not start execution until the branch is resolved, in spite of that it can be issued.

Hardward-based Speculation

- ❑ Speculating on the outcome of branches and executing the program as if the predictions were correct.
  - ➢ Need to handle the situation where the speculation is incorrect.
- ❑ Key ideas:
  - ➢ Dynamic branch prediction to choose which instruction to execute
  - ➢ Speculation to allow the execution of instructions before the control dependences are resolved.
  - ➢ Dynamic scheduling to deal with the scheduling of different combinations of basic blocks.
- ❑ Data flow execution:
  - ➢ Operations execute as soon as their operands are available.

Hardware-Based Speculation

- Execute instructions along predicted execution paths but only commit the results if prediction was correct

- Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative

- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - I.e. updating state or taking an executionb

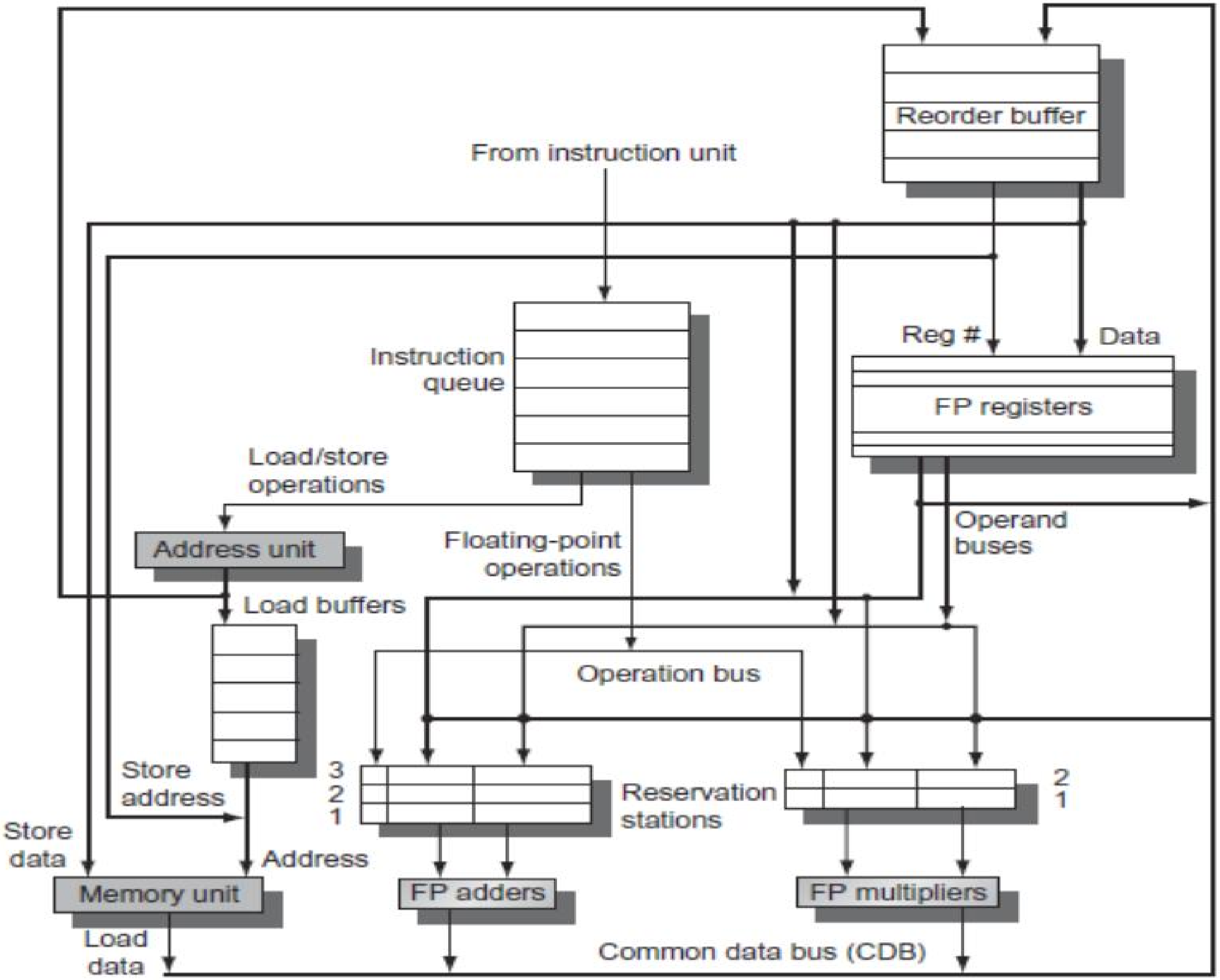
Speculative execution based on Tomasulo's algorithm

- Separate the process of completing execution and the bypassing of results among instructions from instruction commit (reg file or memory update).

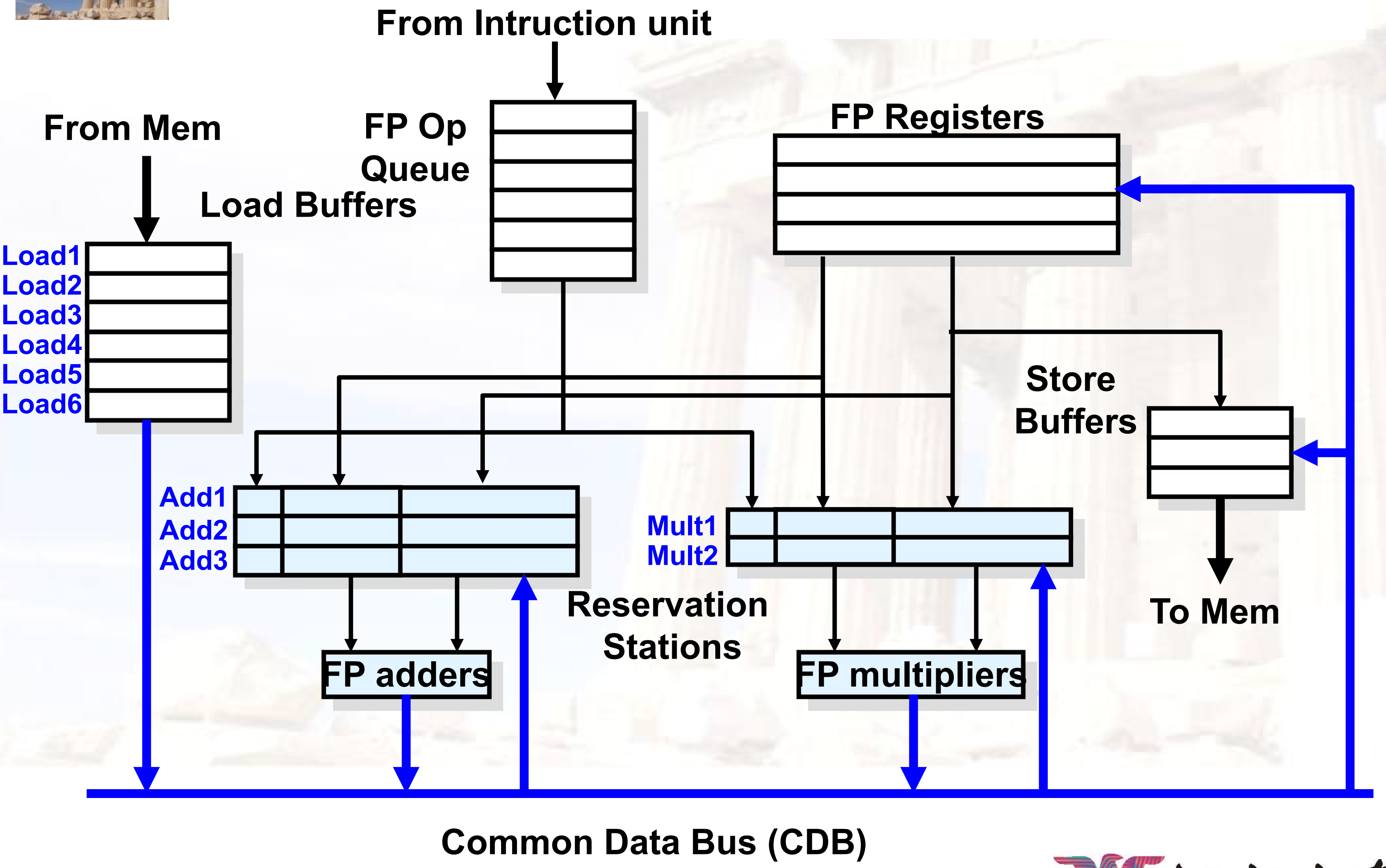
- Allows other (speculative) instructions to execute, but no results are committed until we know the instruction is no longer speculative.

- Allow instructions to execute out of order but force them to commit in order, which helps with handling exceptions properly.

Extend Tomasulo's Algorithm to handle speculation



Review: Tomasulo Algorithm



Difference from Tomasulo Algorithm

- Reorder Buffer is added
- Eliminate store buffer (Mem write is ordered in Reorder buffer)
- Register renaming is done via reorder buffer instead of reservation station
- Reservation only for storing opcodes and operands during the instruction flying between issue and execution.
- ROB hold the instruction results and supply operands in the interval between completion of instruction execution and instruction commit.

Entry of ROB

- Instruction type
- Destination field
- Value field
- Ready field

- Exception vector

4 Steps of 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") , update control entries "in use".

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

3. Write result—finish execution (WB)

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

4. Commit—update register with reorder result

   When instruction 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")

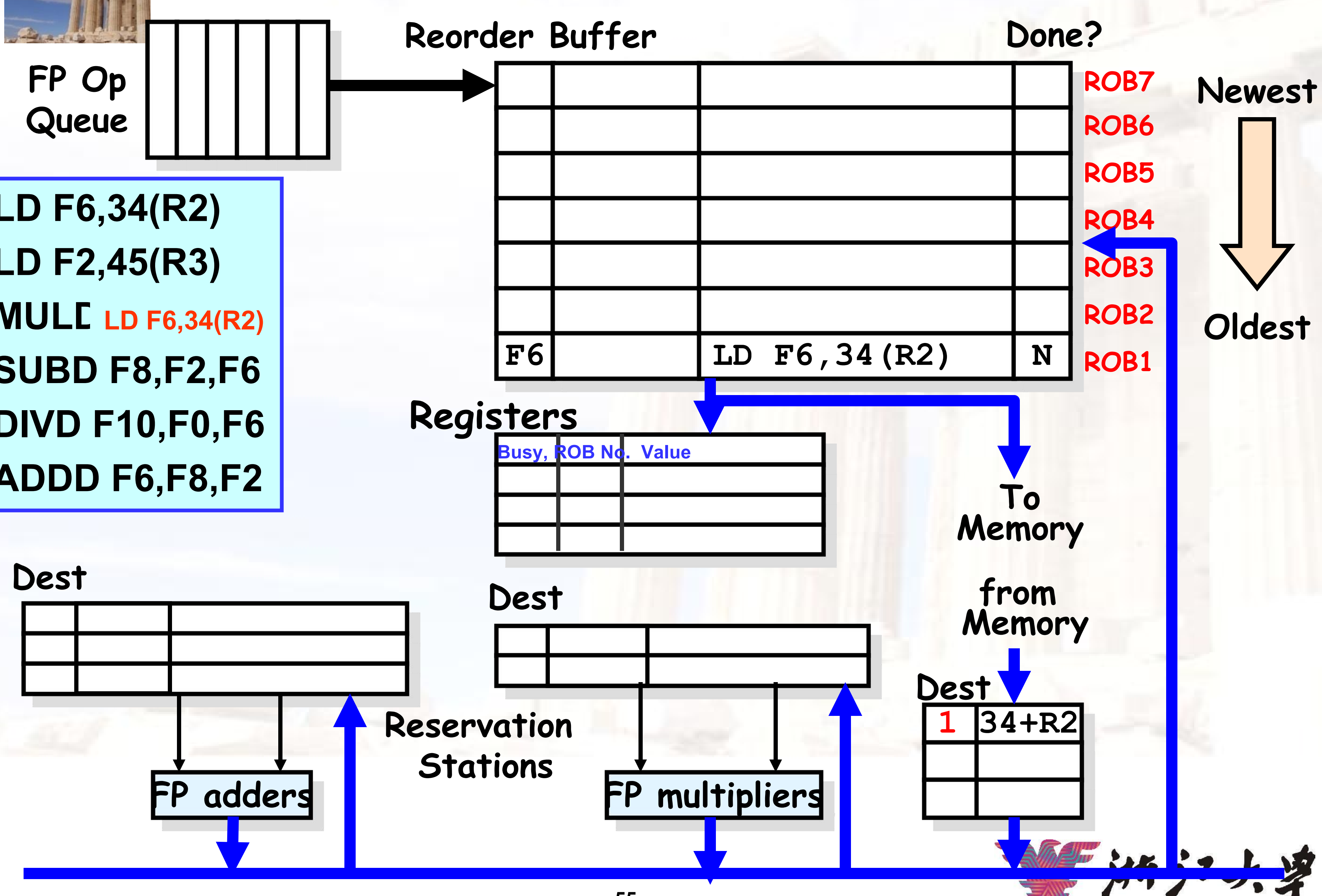
What to do when commit ?

when an instruction reaches the head of ROB and its result is present in the buffer, then

- ❑For non-Branch instruction
  - ➢Update the register with the result
  - ➢Remove the instruction from ROB
  - ➢When it's a STORE, update the memory instead of Reg.
- ❑For a mispredicted branch
  - ➢ROB is flushed and execution is restarted at the correct successor of the branch
- ❑For a right-predicted branch
  - ➢End of branch instruction.

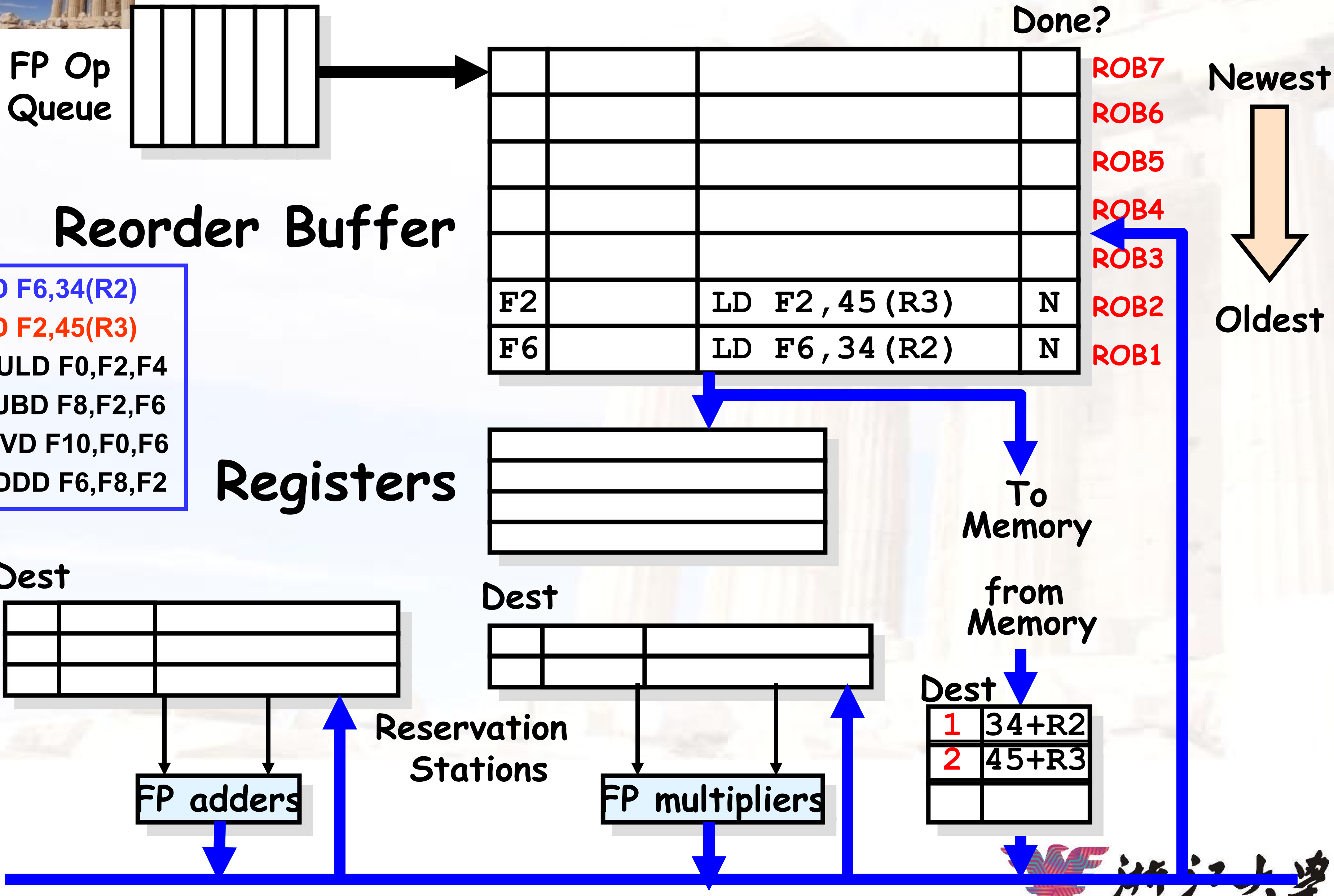
2 clock cycles for pipelined memory access

LD F6,34(R2)
LD F2,45(R3)
MULD LD F6,34(R2)
SUBD F8,F2,F6
DIVD F10,F0,F6
ADDD F6,F8,F2



Tomasulo With Reorder buffer:

- LD F6,34(R2)
- LD F2,45(R3)
- MULD F0,F2,F4
- SUBD F8,F2,F6
- DIVD F10,F0,F6
- ADDD F6,F8,F2

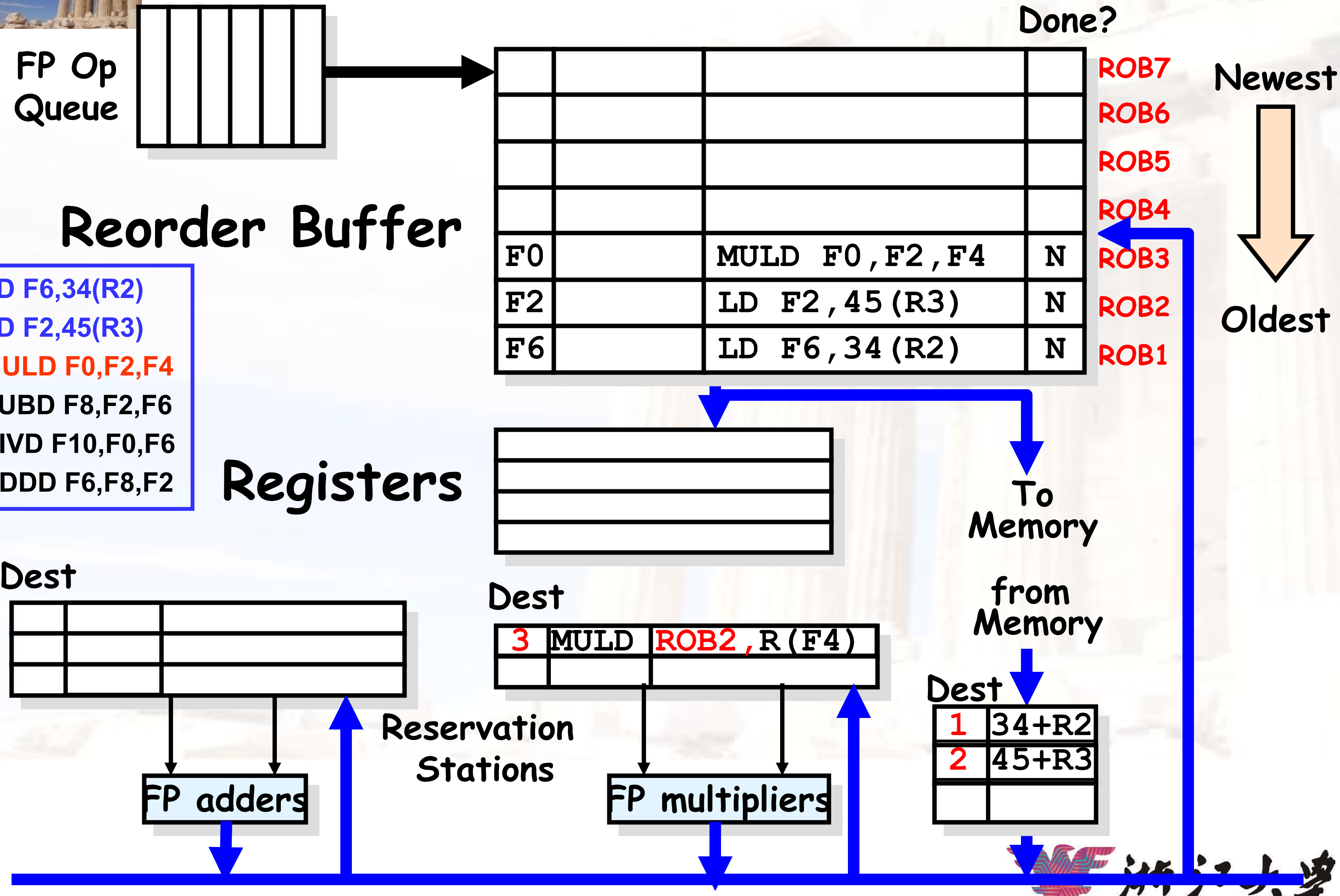


				Done?
ROB7				
ROB6				
ROB5				
ROB4				
ROB3				
ROB2	F2		LD F2,45(R3)	N
ROB1	F6		LD F6,34(R2)	N

Dest	
1	34+R2
2	45+R3

Tomasulo With Reorder buffer:

LD F6,34(R2)
LD F2,45(R3)
MULD F0,F2,F4
SUBD F8,F2,F6
DIVD F10,F0,F6
ADDD F6,F8,F2



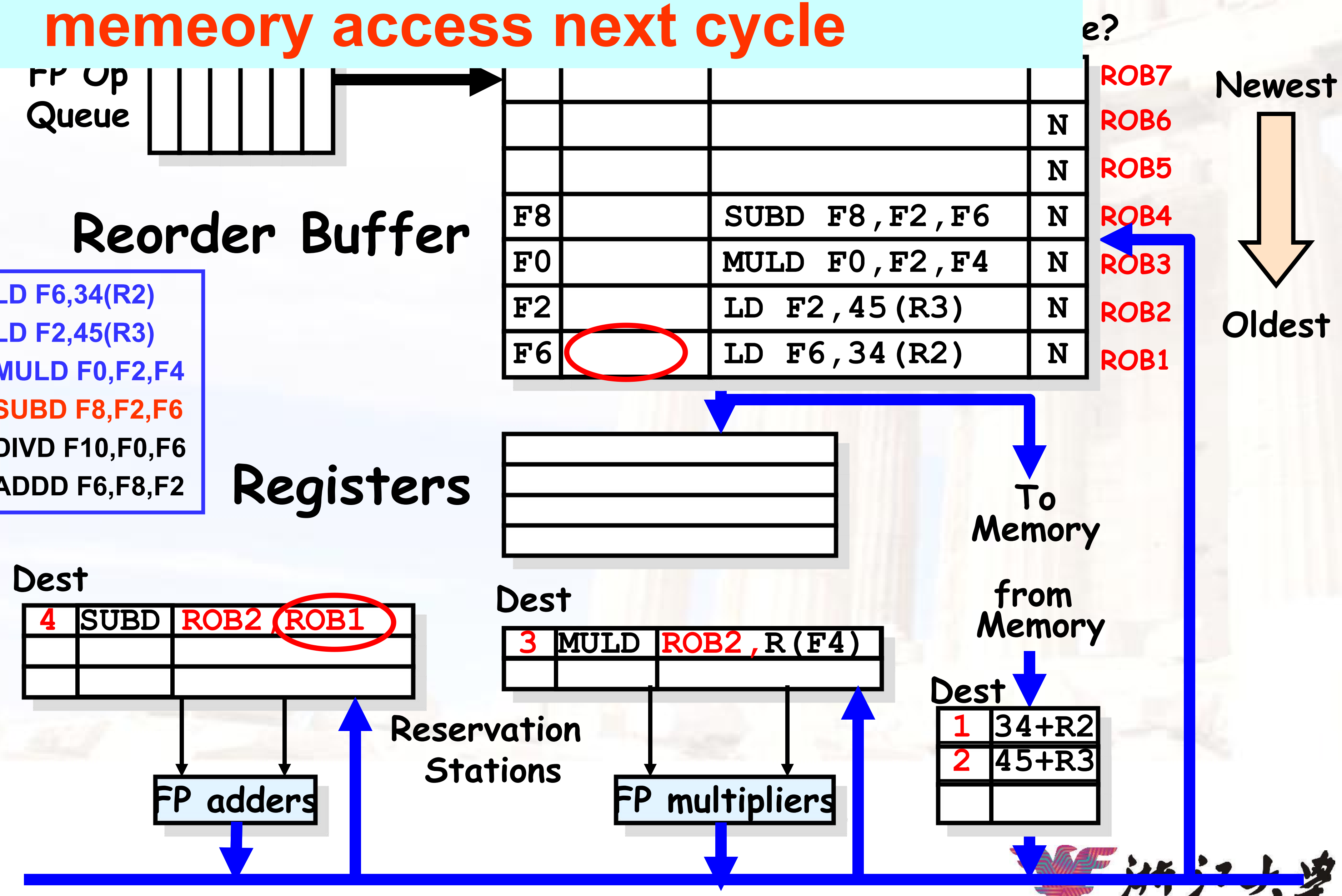
			Done?	
				ROB7
				ROB6
				ROB5
				ROB4
F0		MULD F0,F2,F4	N	ROB3
F2		LD F2,45(R3)	N	ROB2
F6		LD F6,34(R2)	N	ROB1

Dest		
3	MULD	ROB2,R(F4)

Dest	
1	34+R2
2	45+R3

Note: The first LD will finish memeory access next cycle

LD F6,34(R2)
LD F2,45(R3)
MULD F0,F2,F4
SUBD F8,F2,F6
DIVD F10,F0,F6
ADDD F6,F8,F2



Reorder Buffer

				ROB7	Newest
			N	ROB6	
			N	ROB5	
F8		SUBD F8,F2,F6	N	ROB4	
F0		MULD F0,F2,F4	N	ROB3	
F2		LD F2,45(R3)	N	ROB2	
F6		LD F6,34(R2)	N	ROB1	Oldest

Reservation Stations

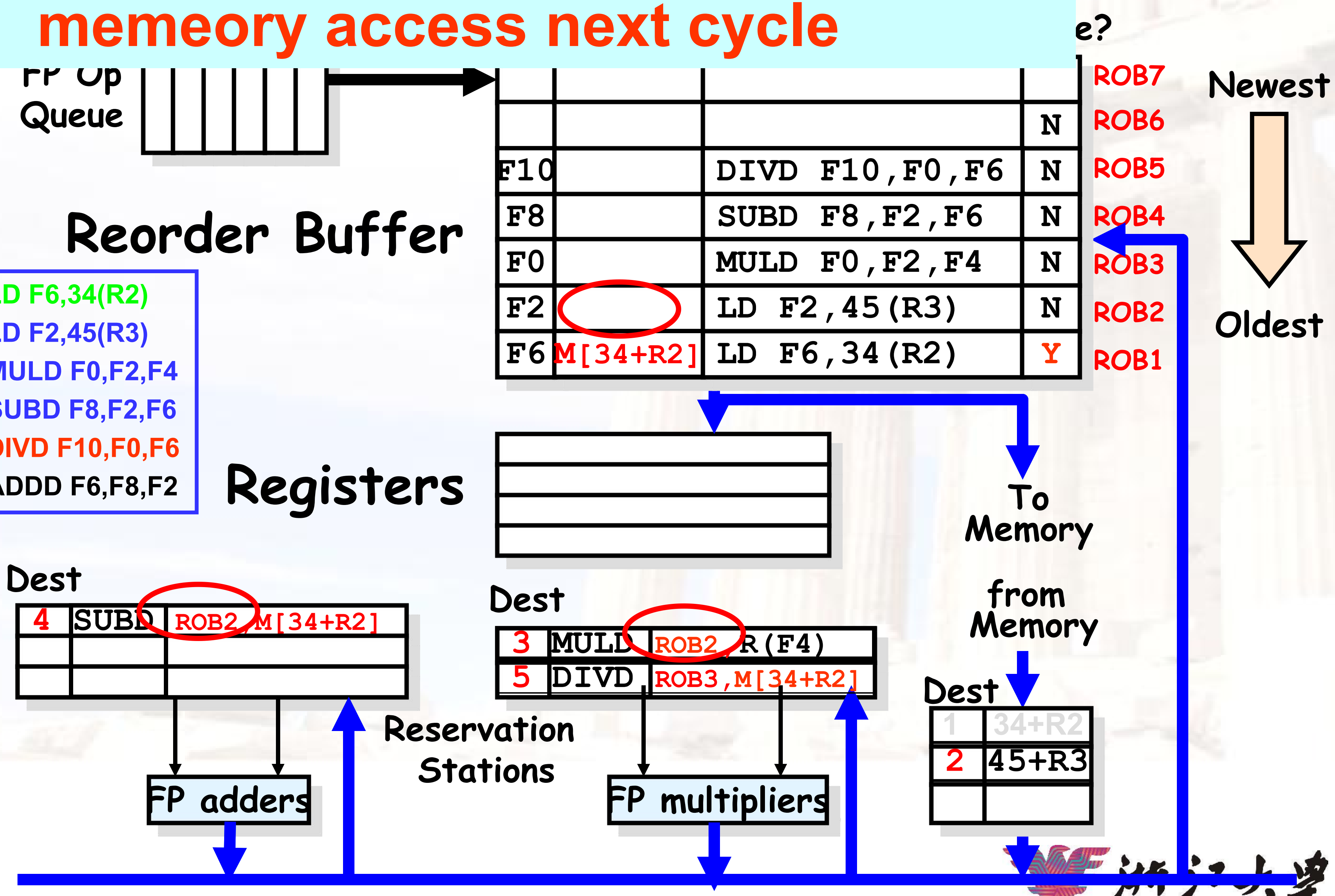
Dest			
4	SUBD	ROB2	ROB1

Dest		
3	MULD	ROB2,R(F4)

Dest	
1	34+R2
2	45+R3

Note: The 2nd LD will finish memeoory access next cycle

LD F6,34(R2)
LD F2,45(R3)
MULD F0,F2,F4
SUBD F8,F2,F6
DIVD F10,F0,F6
ADDD F6,F8,F2



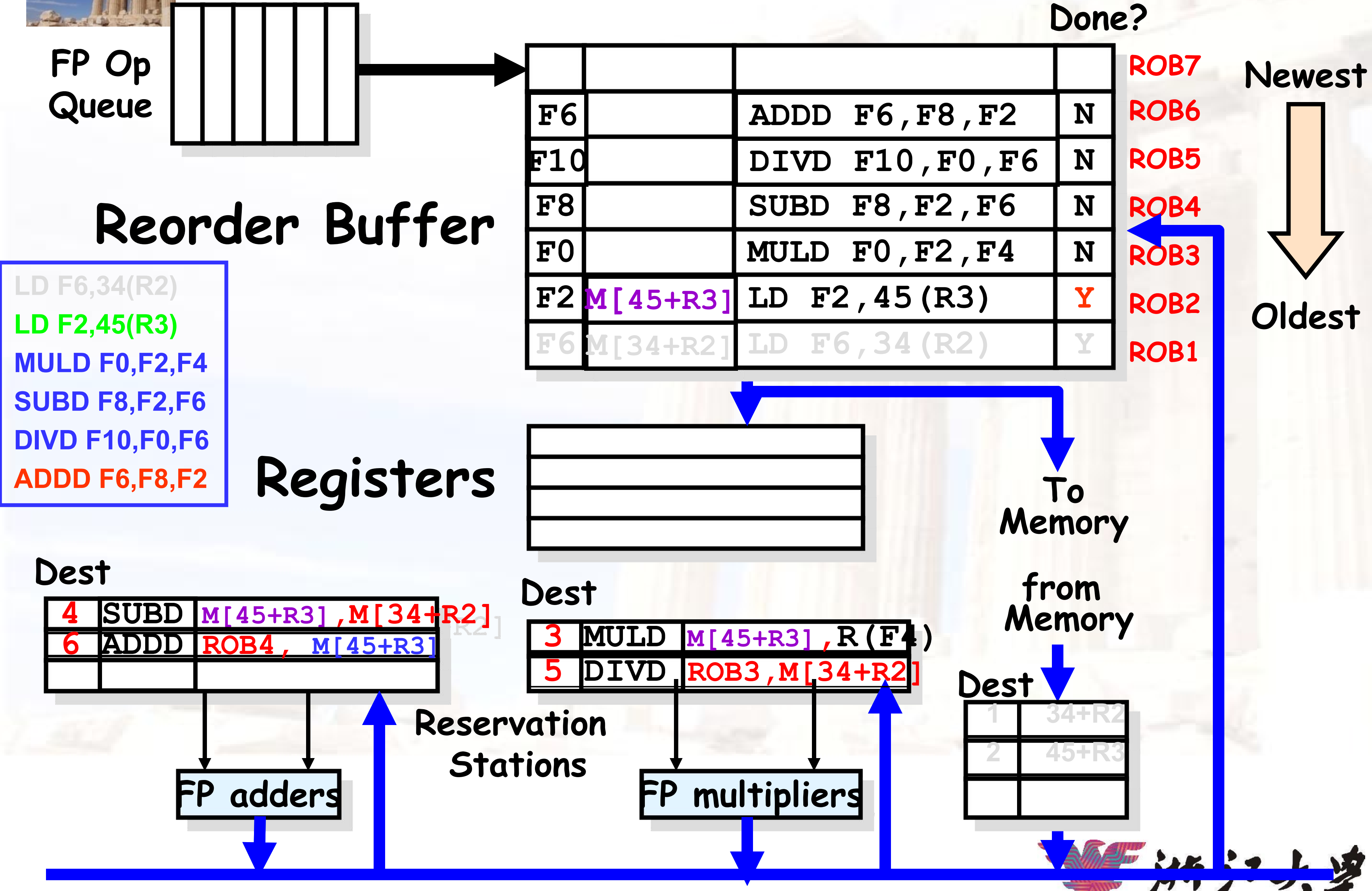
				ROB7
			N	ROB6
F10		DIVD F10,F0,F6	N	ROB5
F8		SUBD F8,F2,F6	N	ROB4
F0		MULD F0,F2,F4	N	ROB3
F2		LD F2,45(R3)	N	ROB2
F6	M[34+R2]	LD F6,34(R2)	Y	ROB1

Dest		
4	SUBD	ROB2,M[34+R2]

Dest		
3	MULD	ROB2,R(F4)
5	DIVD	ROB3,M[34+R2]

Dest	
1	34+R2
2	45+R3

Tomasulo With Reorder buffer:



LD F6,34(R2)
LD F2,45(R3)
MULD F0,F2,F4
SUBD F8,F2,F6
DIVD F10,F0,F6
ADDD F6,F8,F2

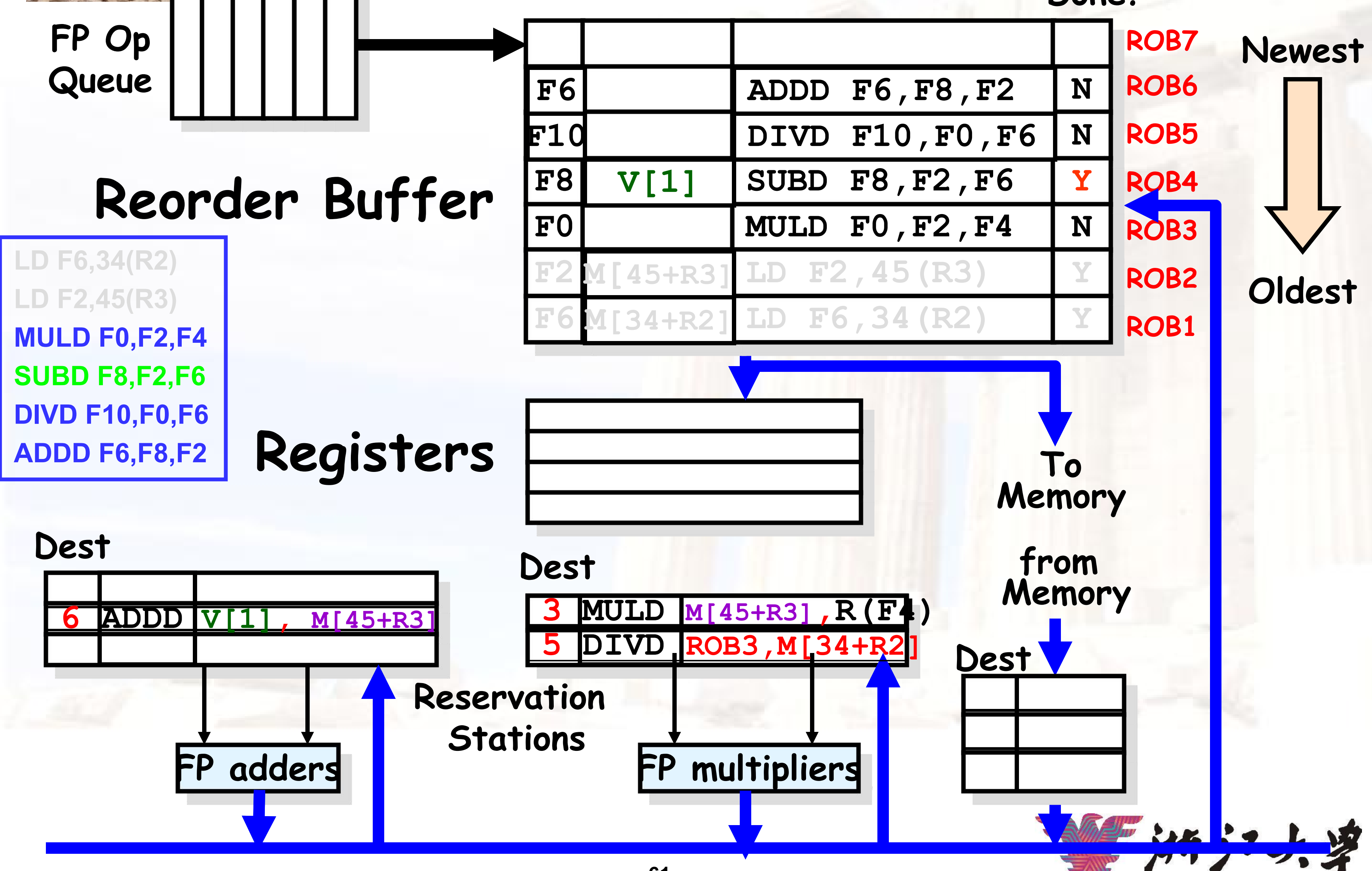
			Done?	
				ROB7
F6		ADDD F6,F8,F2	N	ROB6
F10		DIVD F10,F0,F6	N	ROB5
F8		SUBD F8,F2,F6	N	ROB4
F0		MULD F0,F2,F4	N	ROB3
F2	M[45+R3]	LD F2,45(R3)	Y	ROB2
F6	M[34+R2]	LD F6,34(R2)	Y	ROB1

Dest		
4	SUBD	M[45+R3],M[34+R2]
6	ADDD	ROB4, M[45+R3]

Dest		
3	MULD	M[45+R3],R(F4)
5	DIVD	ROB3,M[34+R2]

Dest	
1	34+R2
2	45+R3

Tomasulo With Reorder buffer:



			Done?	
				ROB7
F6		ADDD F6,F8,F2	N	ROB6
F10		DIVD F10,F0,F6	N	ROB5
F8	V[1]	SUBD F8,F2,F6	Y	ROB4
F0		MULD F0,F2,F4	N	ROB3
F2	M[45+R3]	LD F2,45(R3)	Y	ROB2
F6	M[34+R2]	LD F6,34(R2)	Y	ROB1

LD F6,34(R2)
LD F2,45(R3)
MULD F0,F2,F4
SUBD F8,F2,F6
DIVD F10,F0,F6
ADDD F6,F8,F2

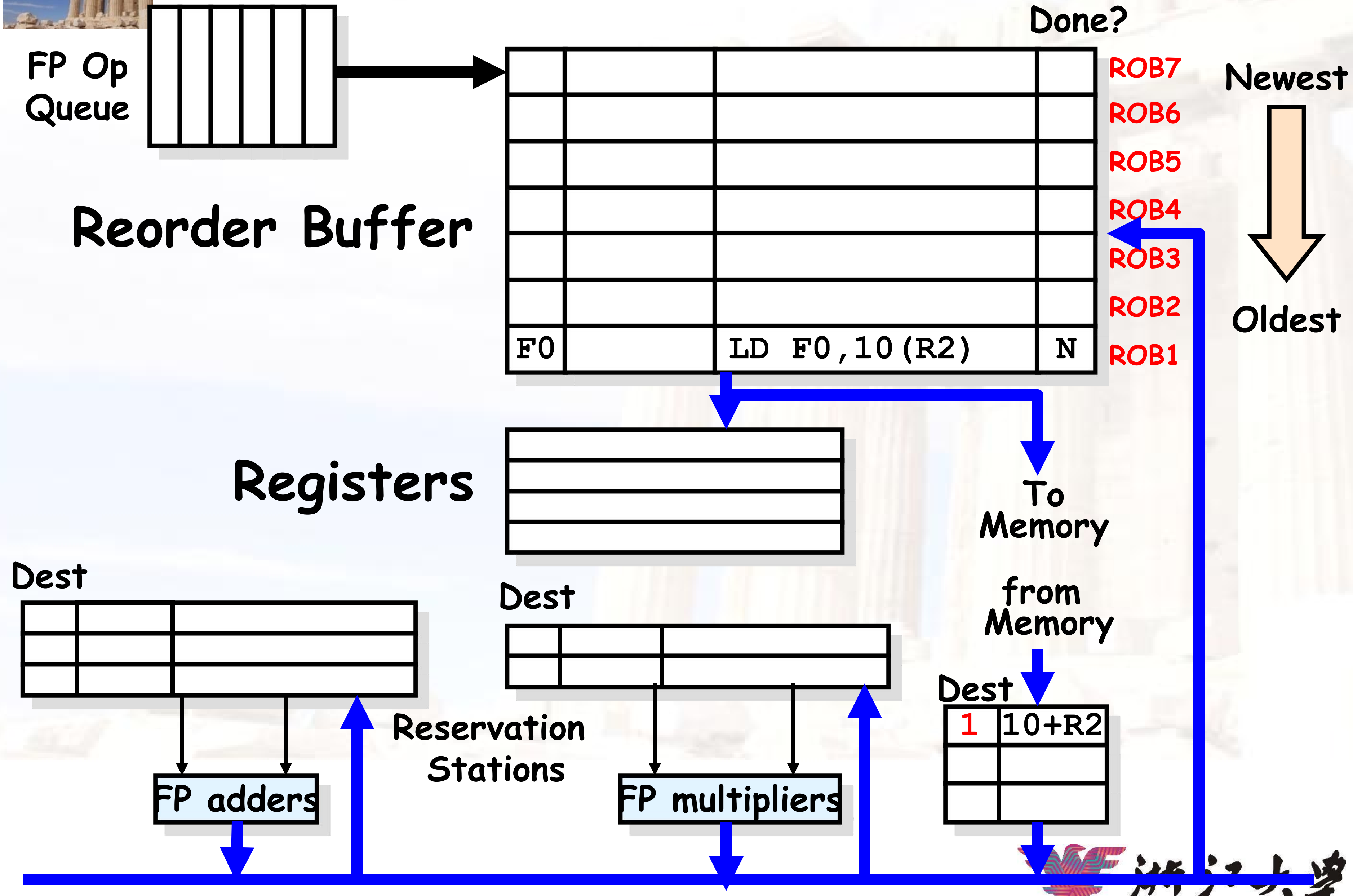
Dest		
6	ADDD	V[1] , M[45+R3]

Dest		
3	MULD	M[45+R3],R(F4)
5	DIVD	ROB3,M[34+R2]

Speculation performance

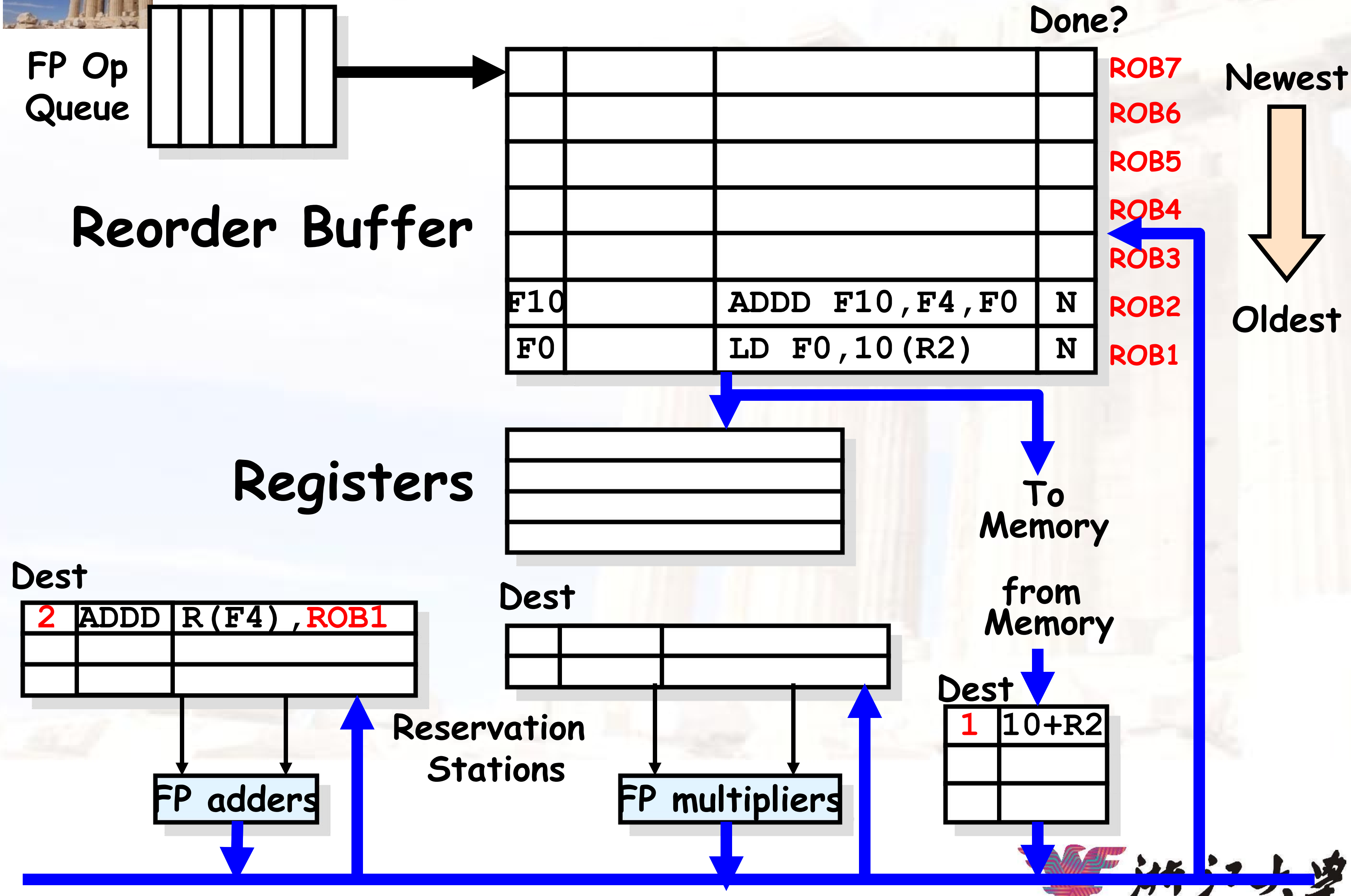
- ❑Overlapped basic block
- ❑Is there any new problems ?
  - ➢Pay attention to the memory accessing

Tomasulo With Reorder buffer:



F0		LD F0,10(R2)	N

Tomasulo With Reorder buffer:

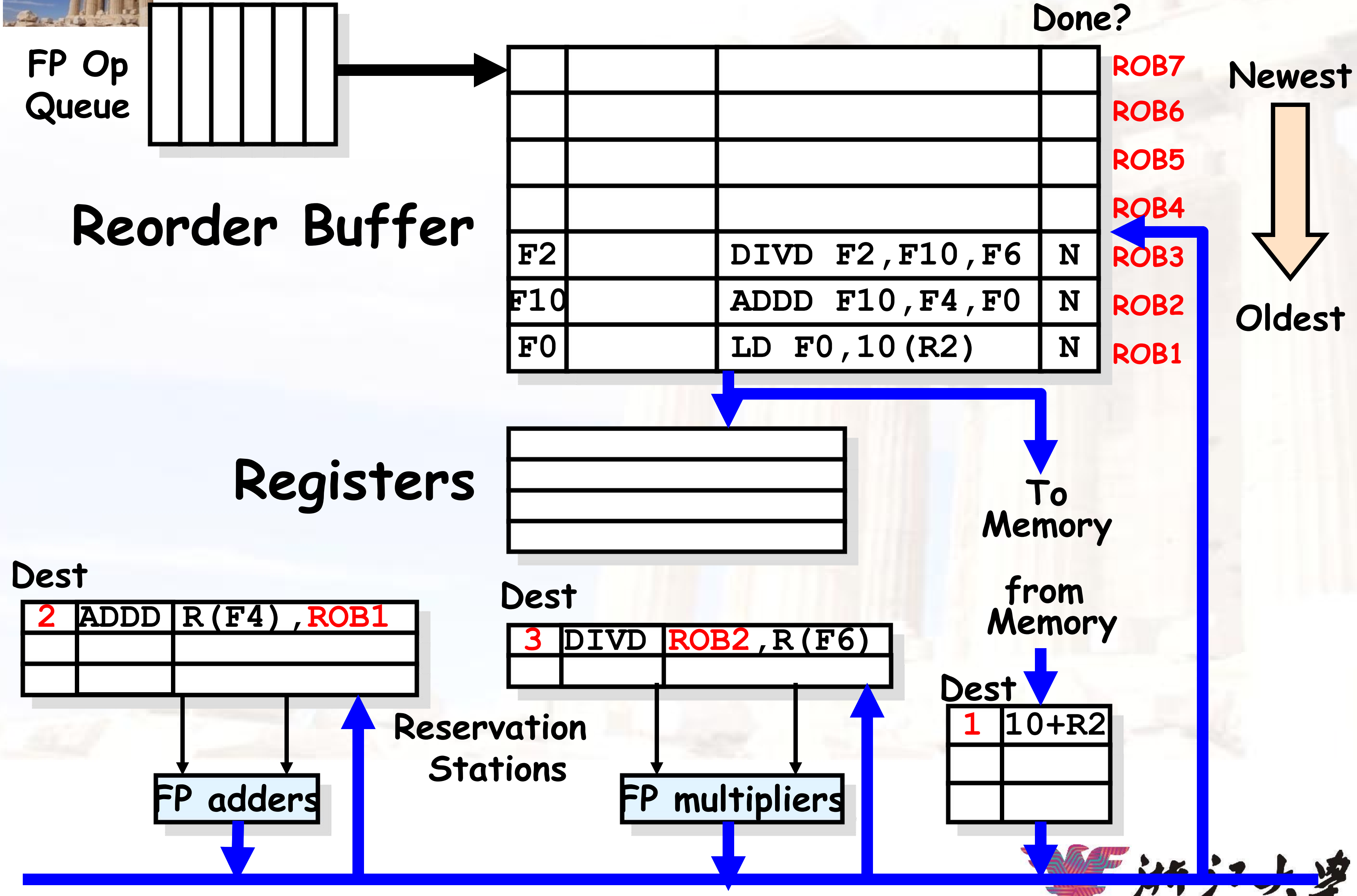


			Done?	
				ROB7
				ROB6
				ROB5
				ROB4
				ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

Dest		
2	ADDD	R(F4),ROB1

Dest	
1	10+R2

Tomasulo With Reorder buffer:



			Done?	
				ROB7
				ROB6
				ROB5
				ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

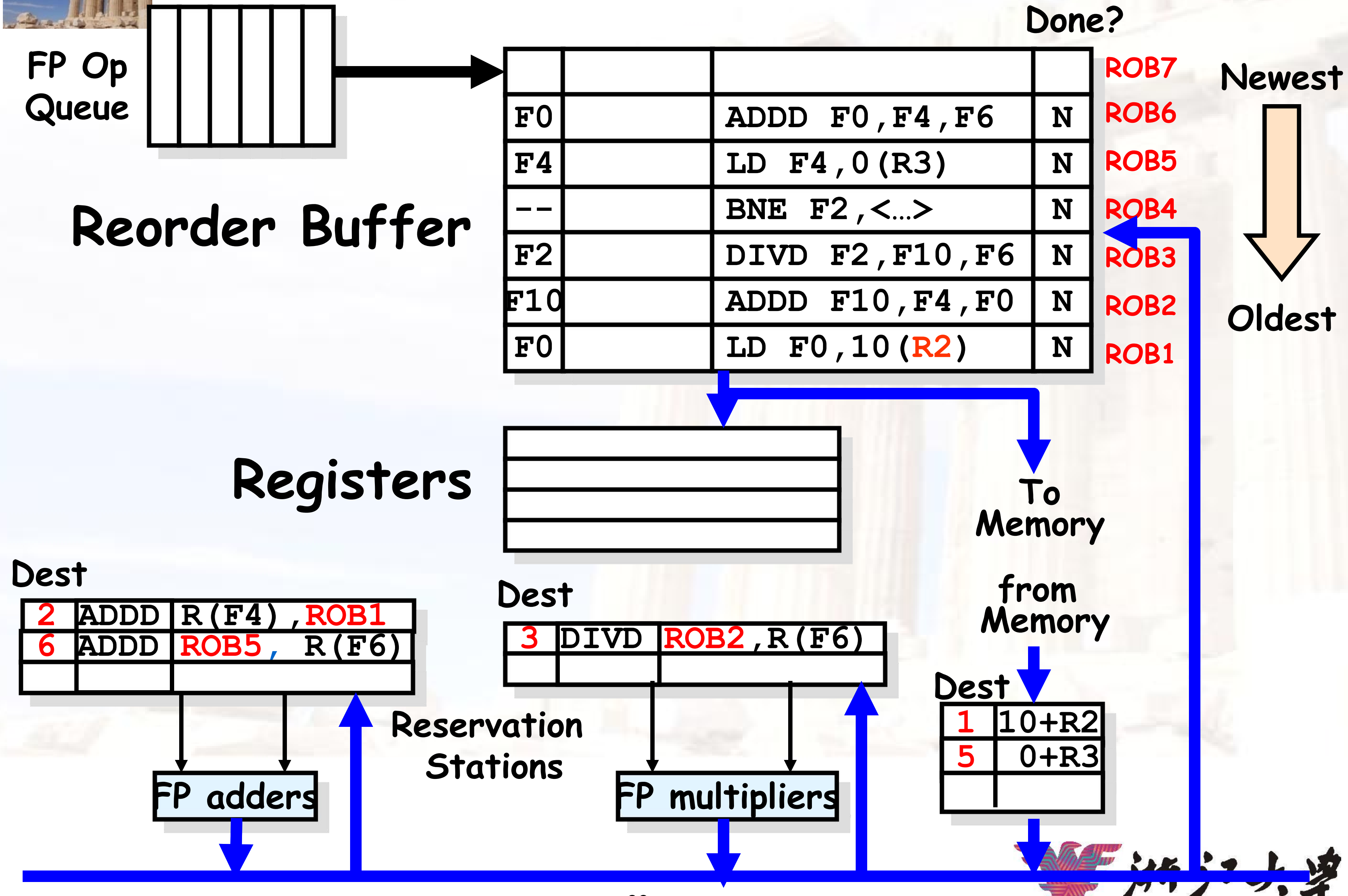
Dest

Dest		
2	ADDD	R(F4),ROB1

Dest		
3	DIVD	ROB2,R(F6)

Dest	
1	10+R2

Tomasulo With Reorder buffer:



			Done?	
				ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4		LD F4,0(R3)	N	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

Dest

2	ADDD	R(F4),ROB1
6	ADDD	ROB5, R(F6)

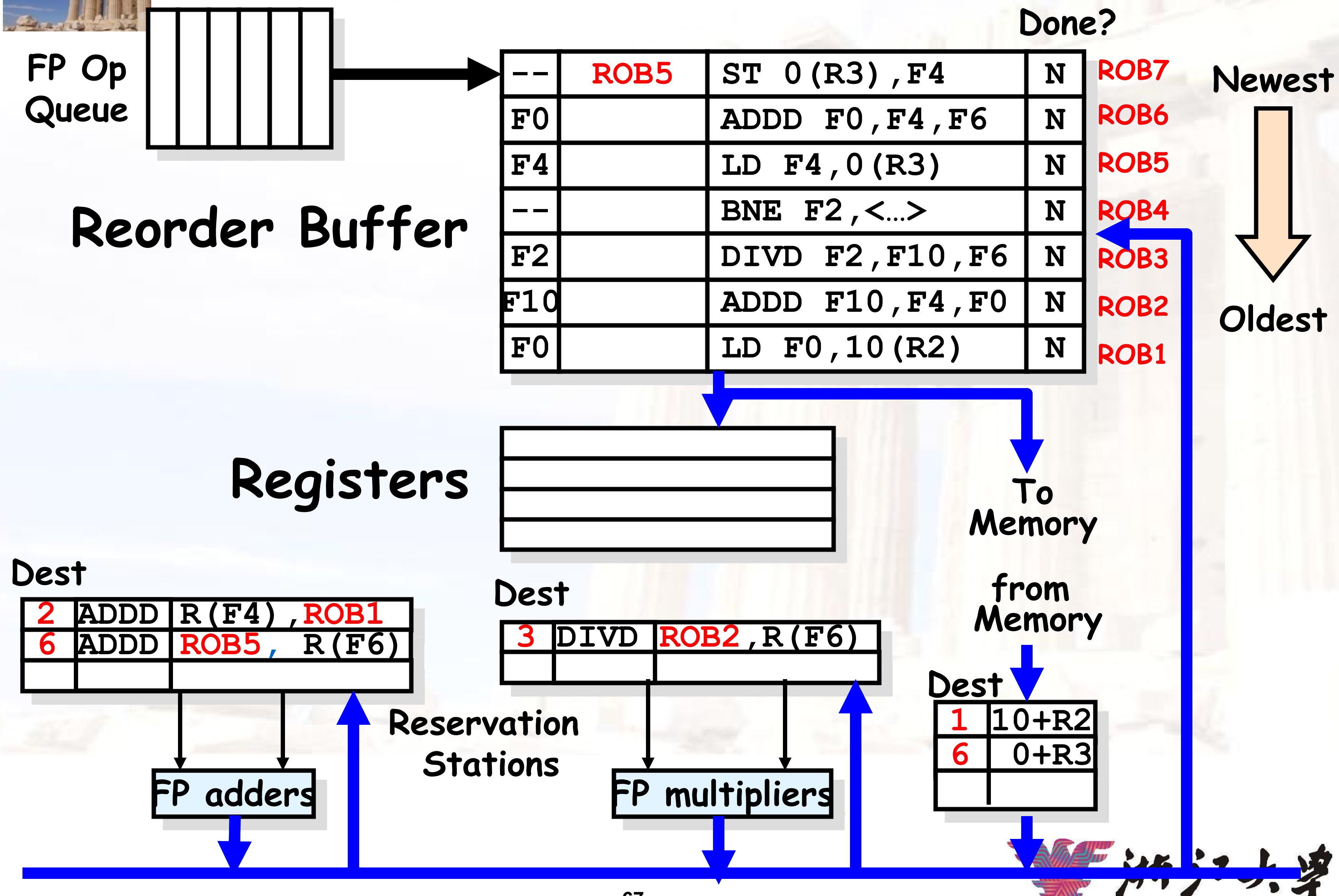
Dest

3	DIVD	ROB2,R(F6)

Dest

1	10+R2
5	0+R3

Tomasulo With Reorder buffer:



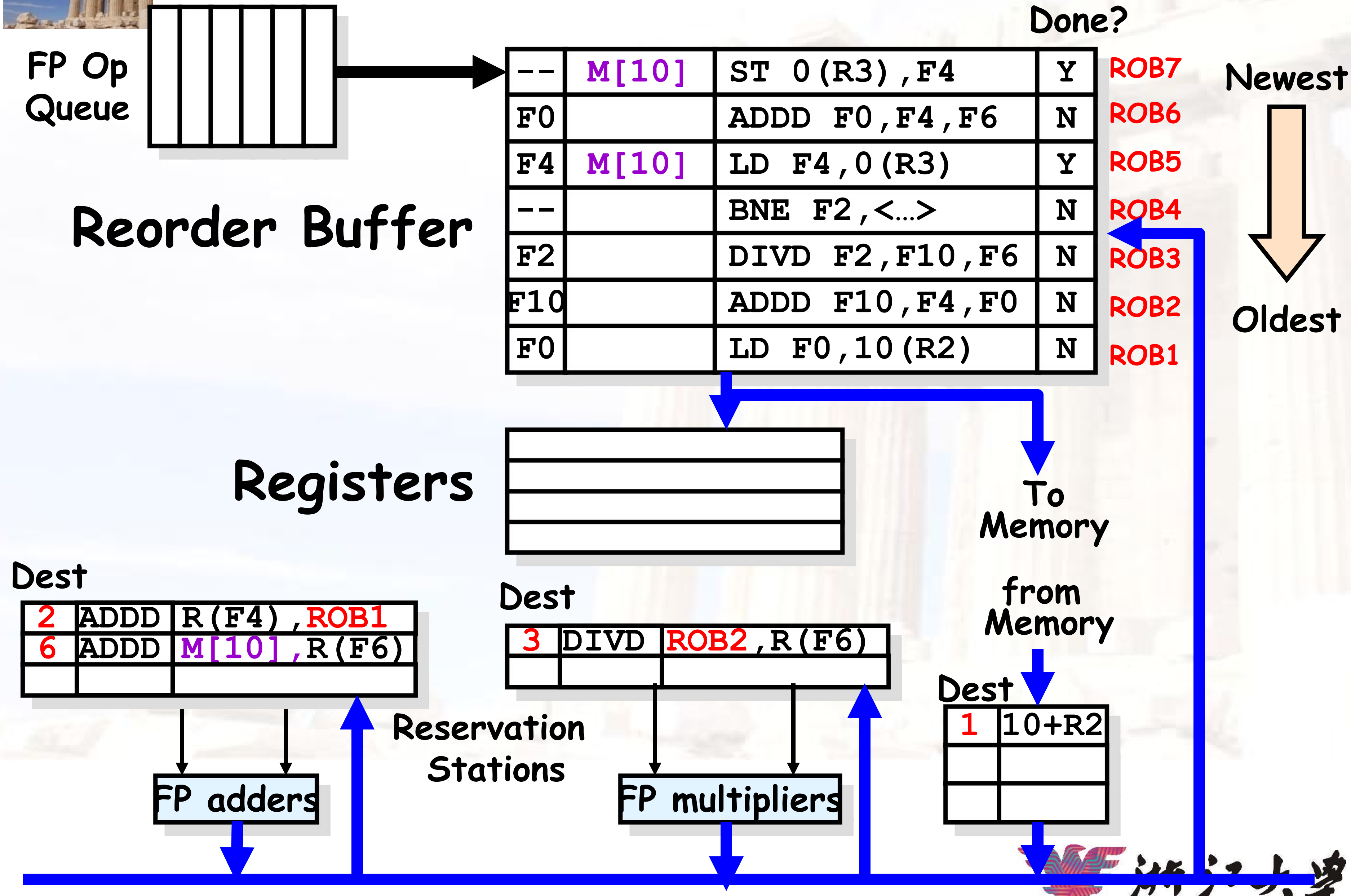
			Done?	
--	ROB5	ST 0(R3),F4	N	ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4		LD F4,0(R3)	N	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

Dest		
2	ADDD	R(F4),ROB1
6	ADDD	ROB5, R(F6)

Dest		
3	DIVD	ROB2,R(F6)

Dest	
1	10+R2
6	0+R3

Tomasulo With Reorder buffer:



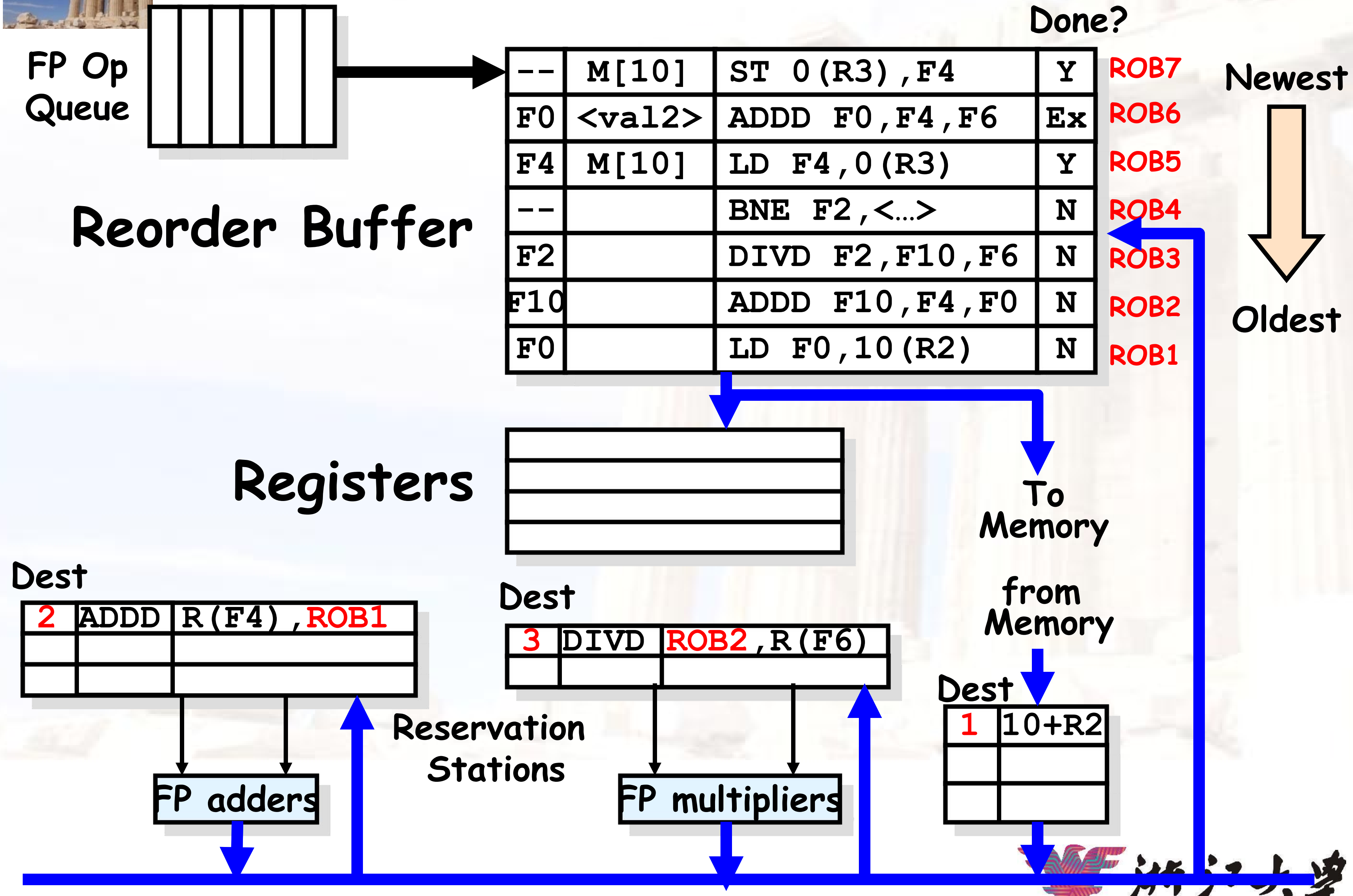
			Done?	
--	M[10]	ST 0(R3),F4	Y	ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4	M[10]	LD F4,0(R3)	Y	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

Dest		
2	ADDD	R(F4),ROB1
6	ADDD	M[10],R(F6)

Dest		
3	DIVD	ROB2,R(F6)

Dest	
1	10+R2

Tomasulo With Reorder buffer:



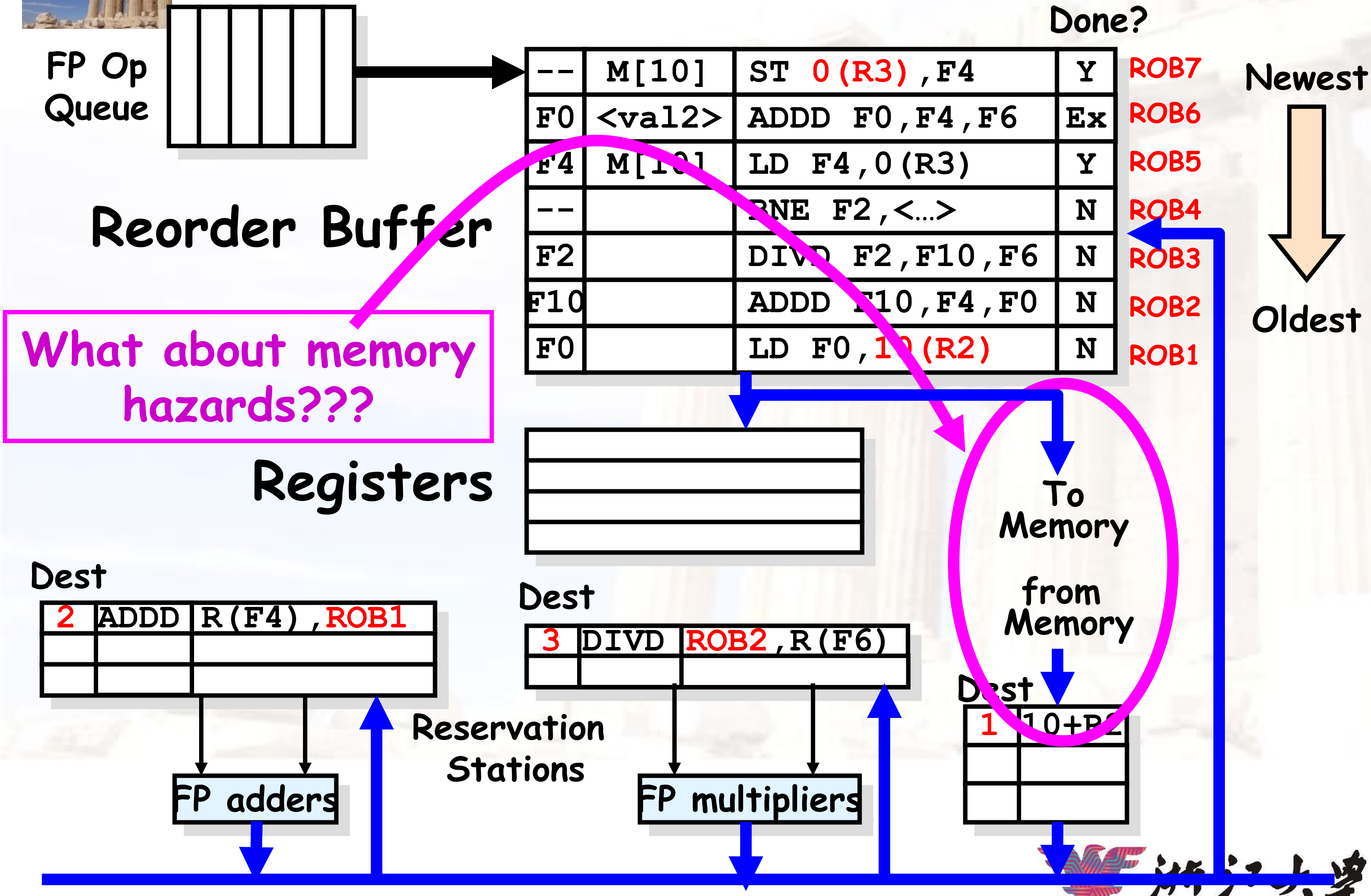
			Done?	
--	M[10]	ST 0(R3),F4	Y	ROB7
F0	<val2>	ADDD F0,F4,F6	Ex	ROB6
F4	M[10]	LD F4,0(R3)	Y	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

Dest		
2	ADDD	R(F4),ROB1

Dest		
3	DIVD	ROB2,R(F6)

Dest	
1	10+R2

Tomasulo With Reorder buffer:



			Done?	
--	M[10]	ST 0(R3),F4	Y	ROB7
F0	<val2>	ADDD F0,F4,F6	Ex	ROB6
F4	M[10]	LD F4,0(R3)	Y	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

What about memory hazards???

Dest		
2	ADDD	R(F4),ROB1

Dest		
3	DIVD	ROB2,R(F6)

Dest	
1	10+R2

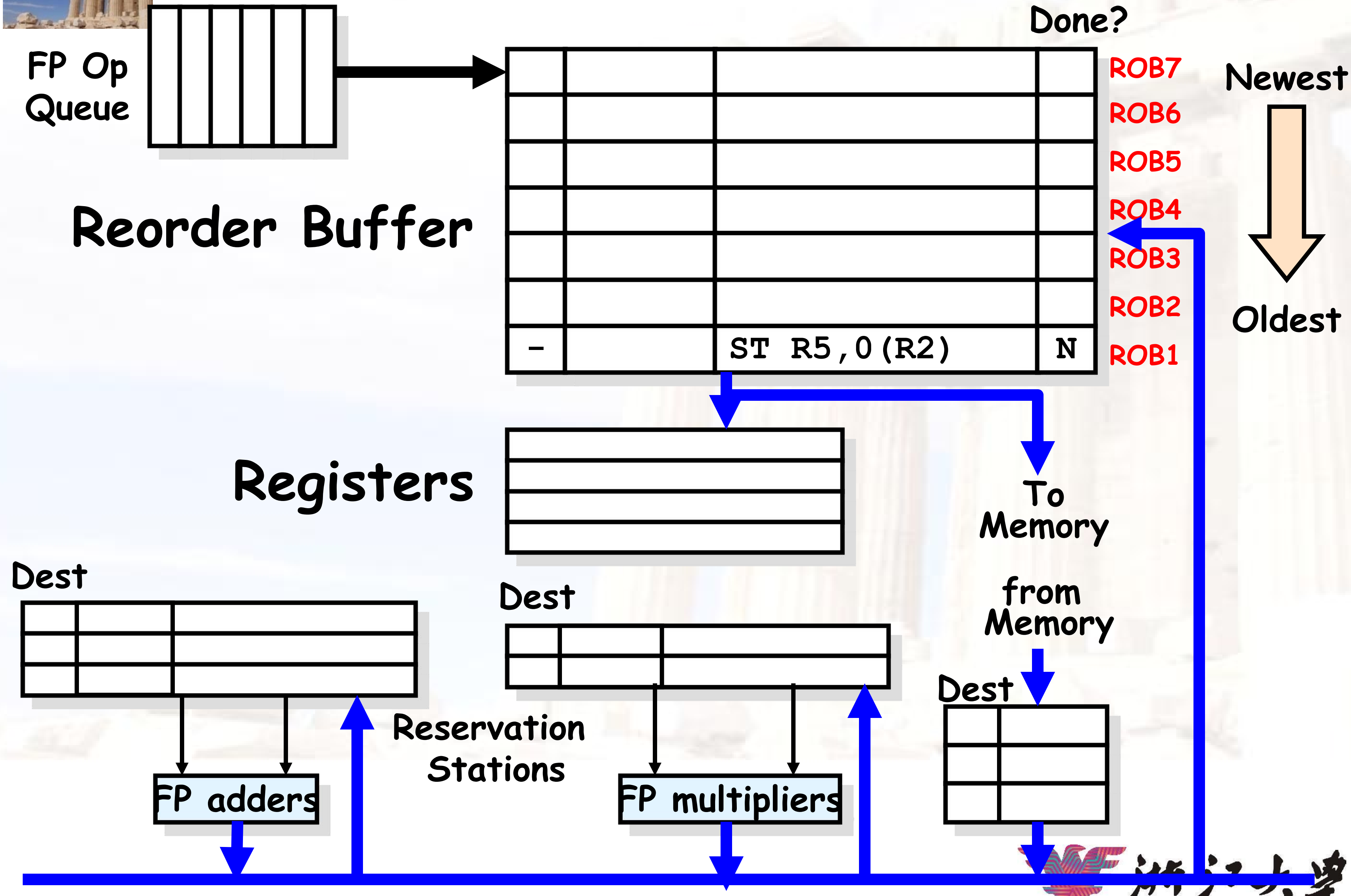
new problem--memory disambiguatio

- ❑Question: Given a load that follows a store in program order, are the two related?
  - ➢(Alternatively: is there a RAW hazard between the store and the load)?

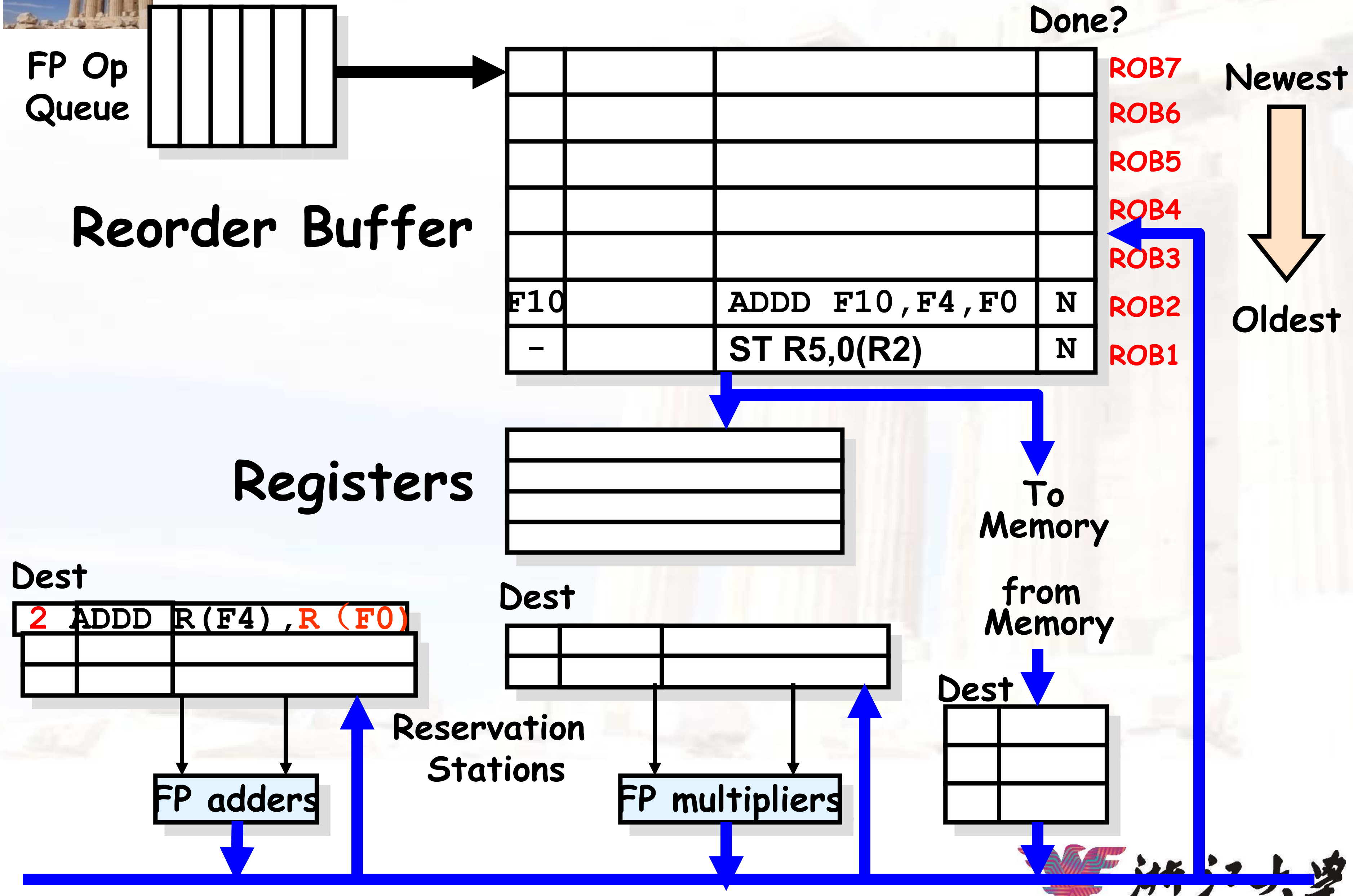
```
Eg:        ST     0(R2),  R5
           LD     R4,     0(R3)
```

- ❑Can we go ahead and start the load early?
  - ➢Store address could be delayed for a long time by some calculation that leads to R2 (divide?).
  - ➢We might want to issue/begin execution of both operations in same cycle.

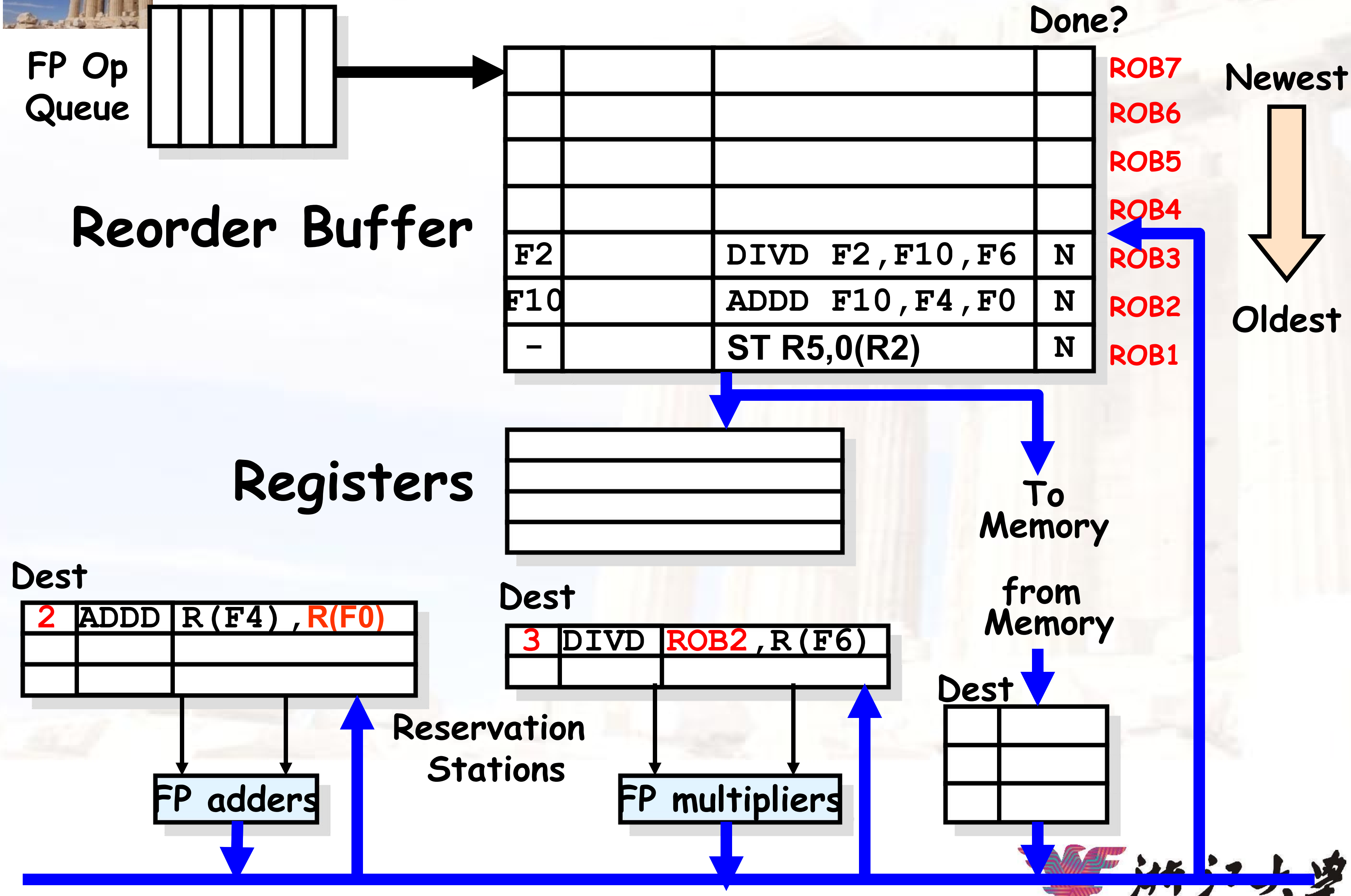
Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:

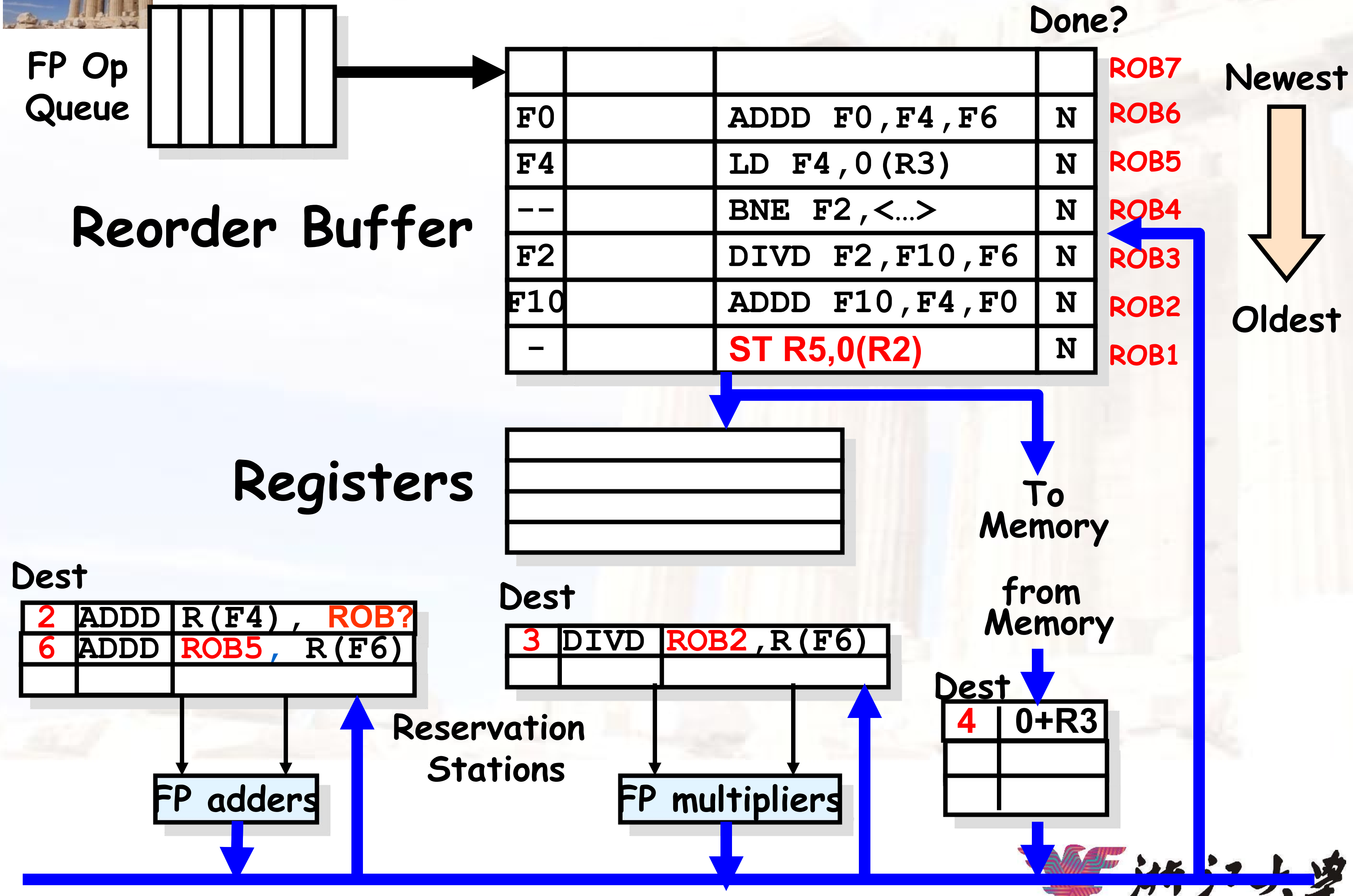


				Done?
ROB7				
ROB6				
ROB5				
ROB4				
ROB3	F2		DIVD F2,F10,F6	N
ROB2	F10		ADDD F10,F4,F0	N
ROB1	-		ST R5,0(R2)	N

Dest		
2	ADDD	R(F4),R(F0)

Dest		
3	DIVD	ROB2,R(F6)

Tomasulo With Reorder buffer:



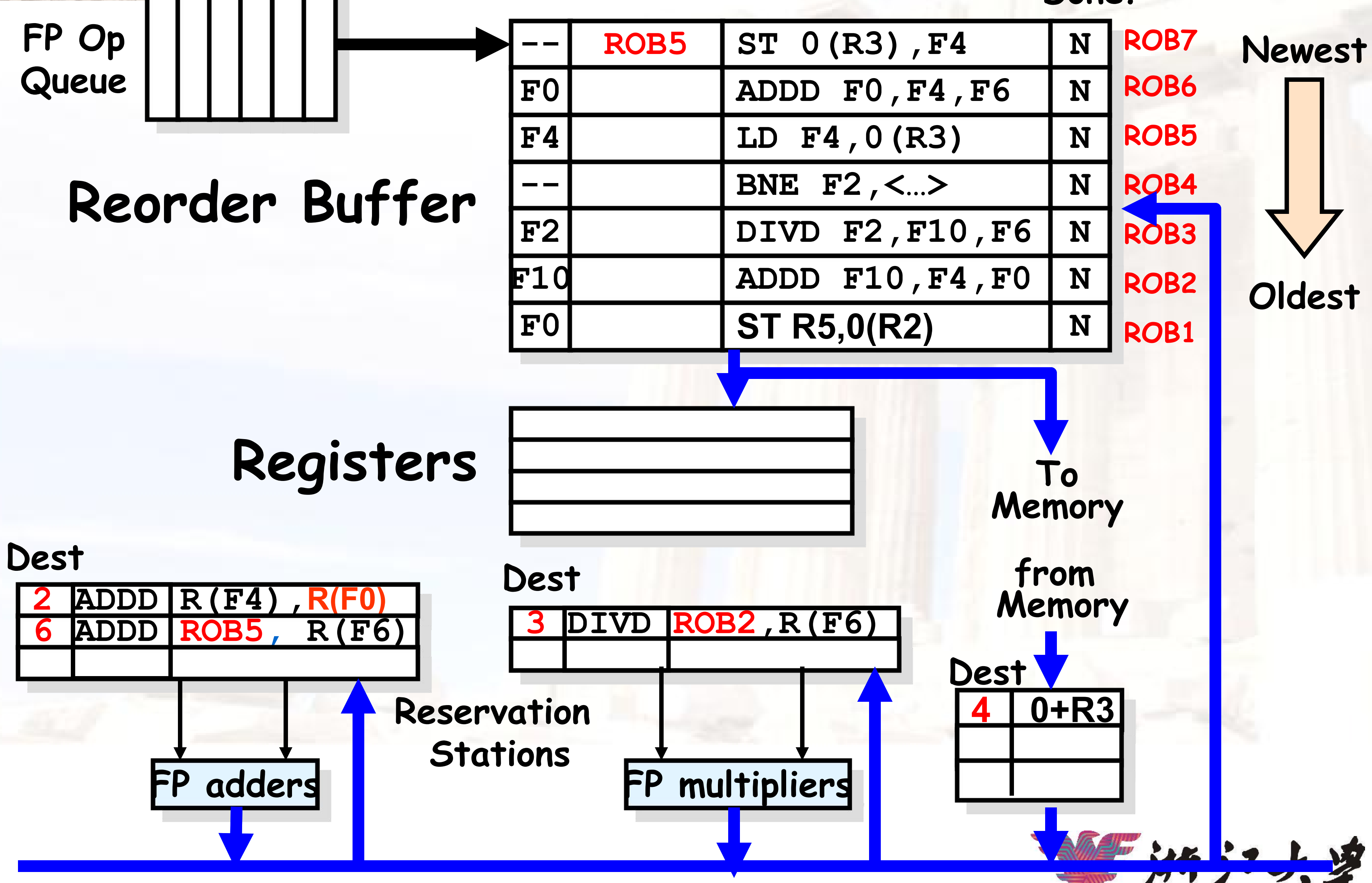
			Done?	
				ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4		LD F4,0(R3)	N	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
-		ST R5,0(R2)	N	ROB1

Dest		
2	ADDD	R(F4), ROB?
6	ADDD	ROB5, R(F6)

Dest		
3	DIVD	ROB2,R(F6)

Dest	
4	0+R3

Tomasulo With Reorder buffer:



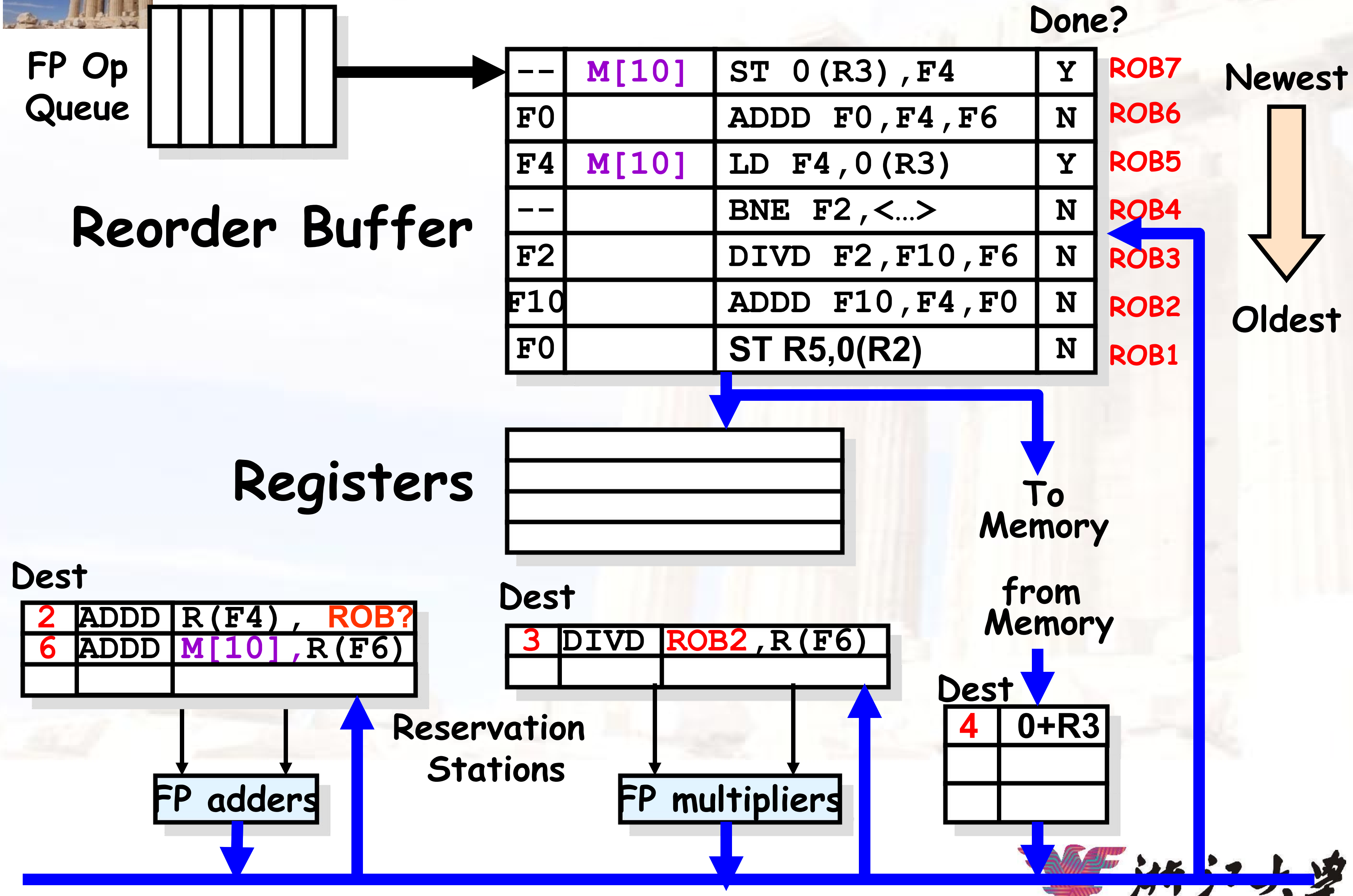
			Done?	
--	ROB5	ST 0(R3),F4	N	ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4		LD F4,0(R3)	N	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		ST R5,0(R2)	N	ROB1

Dest		
2	ADDD	R(F4),R(F0)
6	ADDD	ROB5, R(F6)

Dest		
3	DIVD	ROB2,R(F6)

Dest	
4	0+R3

Tomasulo With Reorder buffer:



FP Op Queue

Reorder Buffer

			Done?	
--	M[10]	ST 0(R3),F4	Y	ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4	M[10]	LD F4,0(R3)	Y	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		ST R5,0(R2)	N	ROB1

Newest

Oldest

Registers

To Memory

from Memory

Reservation Stations

Dest		
2	ADDD	R(F4), ROB?
6	ADDD	M[10],R(F6)

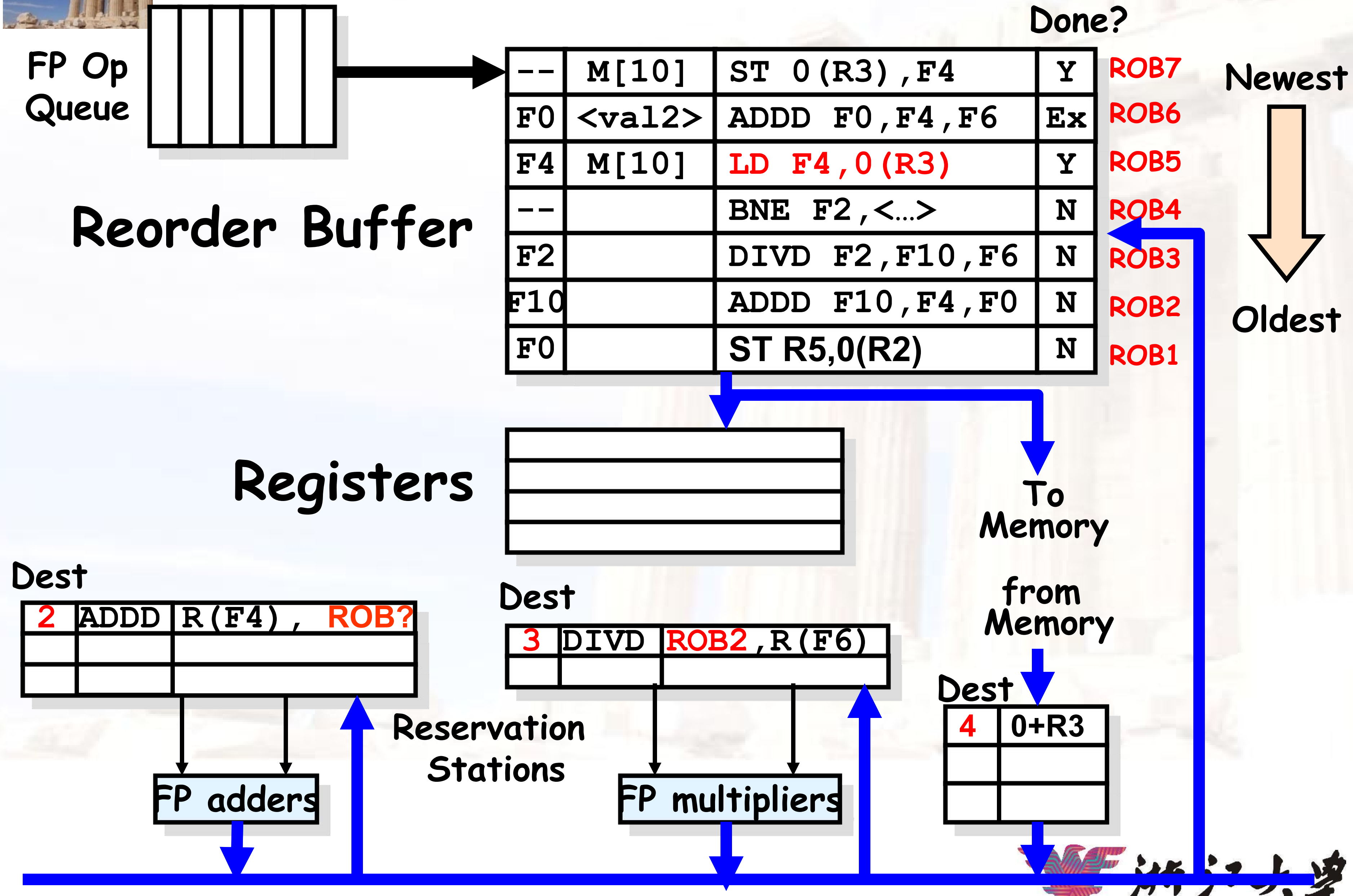
FP adders

Dest		
3	DIVD	ROB2,R(F6)

FP multipliers

Dest	
4	0+R3

Tomasulo With Reorder buffer:



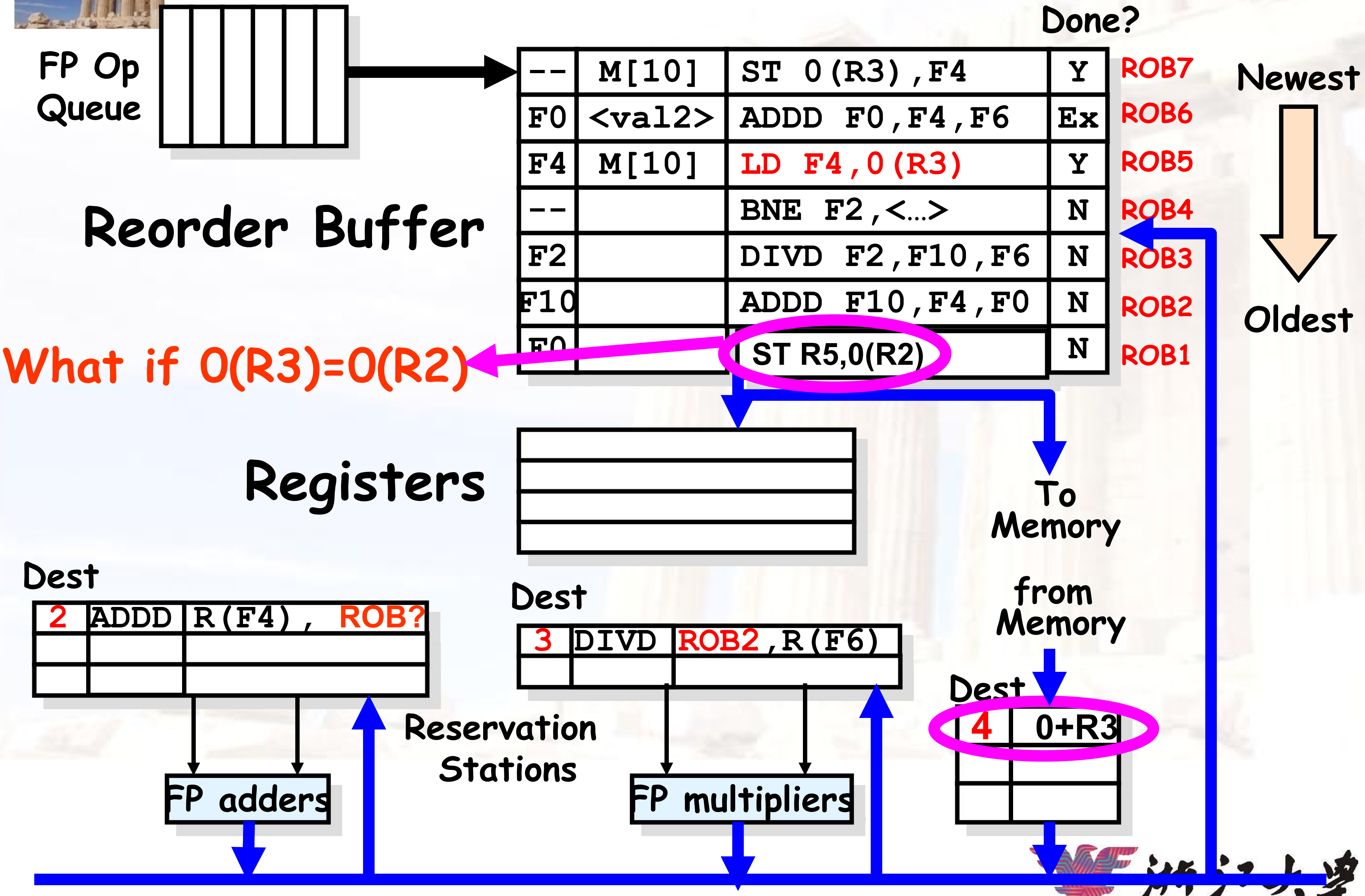
			Done?	
--	M[10]	ST 0(R3),F4	Y	ROB7
F0	<val2>	ADDD F0,F4,F6	Ex	ROB6
F4	M[10]	LD F4,0(R3)	Y	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		ST R5,0(R2)	N	ROB1

Dest		
2	ADDD	R(F4), ROB?

Dest		
3	DIVD	ROB2,R(F6)

Dest	
4	0+R3

Tomasulo With Reorder buffer:



			Done?	
--	M[10]	ST 0(R3),F4	Y	ROB7
F0	<val2>	ADDD F0,F4,F6	Ex	ROB6
F4	M[10]	LD F4,0(R3)	Y	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		ST R5,0(R2)	N	ROB1

Dest		
2	ADDD	R(F4), ROB?

Dest		
3	DIVD	ROB2,R(F6)

Dest	
4	0+R3

Memory Disambiguation: Sorting out RAW Hazards in memory

- Question: Given a load that follows a store in program order, are the two related?
  - (Alternatively: is there a RAW hazard between the store and the load)?

```
Eg:     ST      0(R2),   R5
        LD      R6,      0(R3)
```

- Can we go ahead and start the load early?
  - Store address could be delayed for a long time by some calculation that leads to R2 (divide?).
  - We might want to issue/begin execution of both operations in same cycle.
  - Today: Answer is that we are not allowed to start load until we know that address 0(R2) ≠ 0(R3)

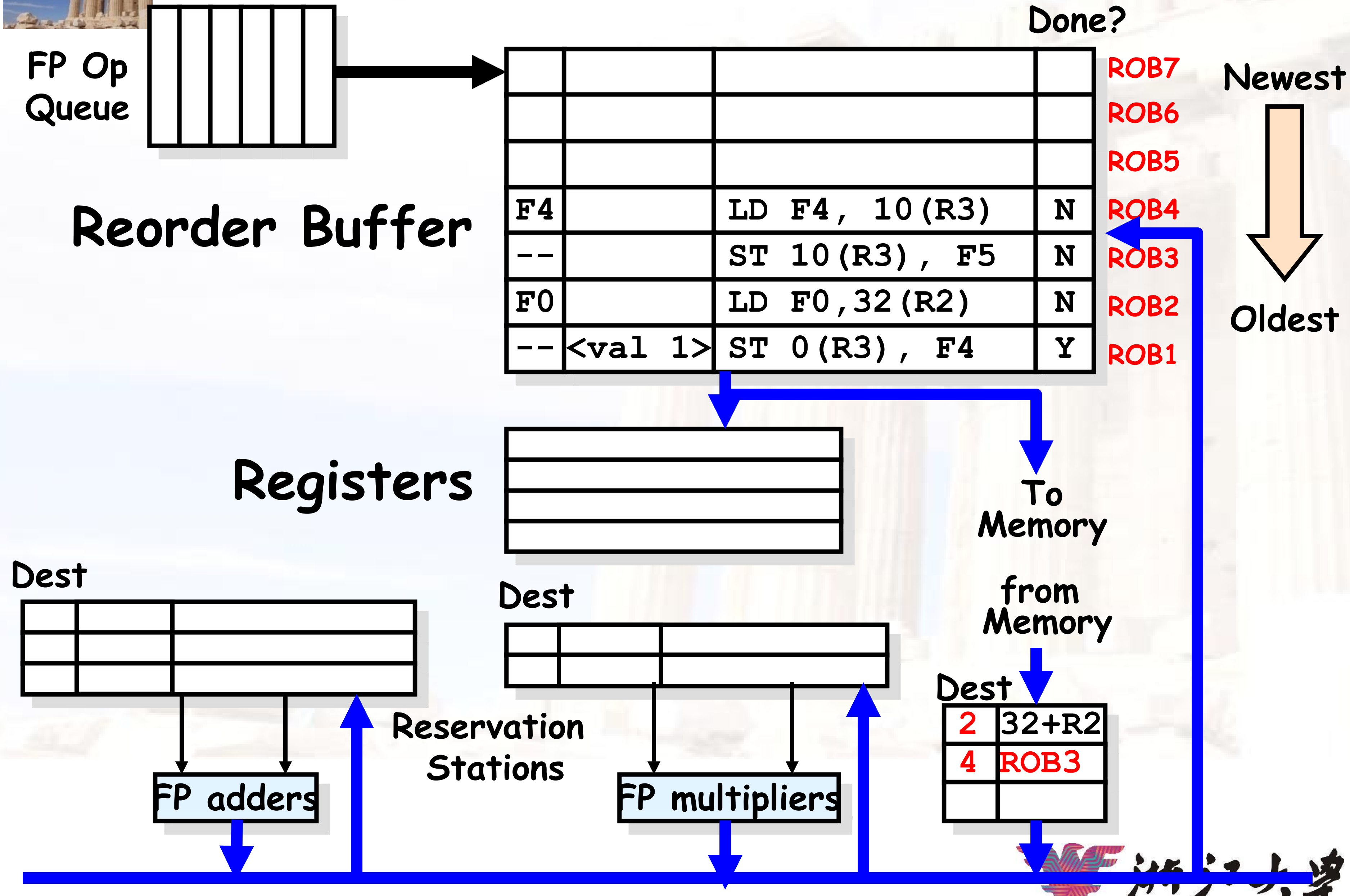
Hardware Support for Memory Disambiguation

- Need buffer to keep track of all outstanding stores to memory, in program order.
  - Keep track of address (when becomes available) and value (when becomes available)
  - FIFO ordering: will retire stores from this buffer in program order
- When issuing a load, record current head of store queue (know which stores are ahead of you).

Hardware Support for Memory Disambiguation(2)

- ❑ When have address for load, check store queue:
  - ➤ If *any* store prior to load is waiting for its address, stall load.
  - ➤ If load address matches earlier store address (associative lookup), then we have a *memory-induced RAW hazard*:
    - ○ store value available ⇒ return value
    - ○ store value not available ⇒ return ROB number of source
  - ➤ Otherwise, send out request to memory
- ❑ Actual stores commit in order, so no worry about WAR/WAW hazards through memory.

Memory Disambiguation:

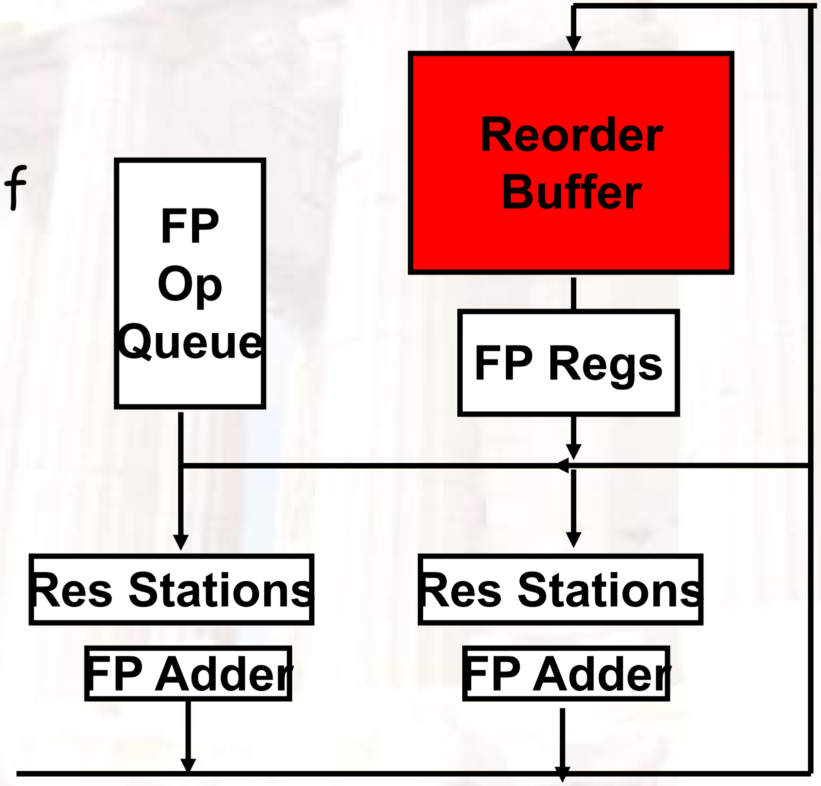


			Done?	
				ROB7
				ROB6
				ROB5
F4		LD F4, 10(R3)	N	ROB4
--		ST 10(R3), F5	N	ROB3
F0		LD F0,32(R2)	N	ROB2
--	<val 1>	ST 0(R3), F4	Y	ROB1

Dest	
2	32+R2
4	ROB3

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)
  - Once instruction commits, result is put into register
  - As a result, easy to undo speculated instructions
    on mispredicted branches or exceptions



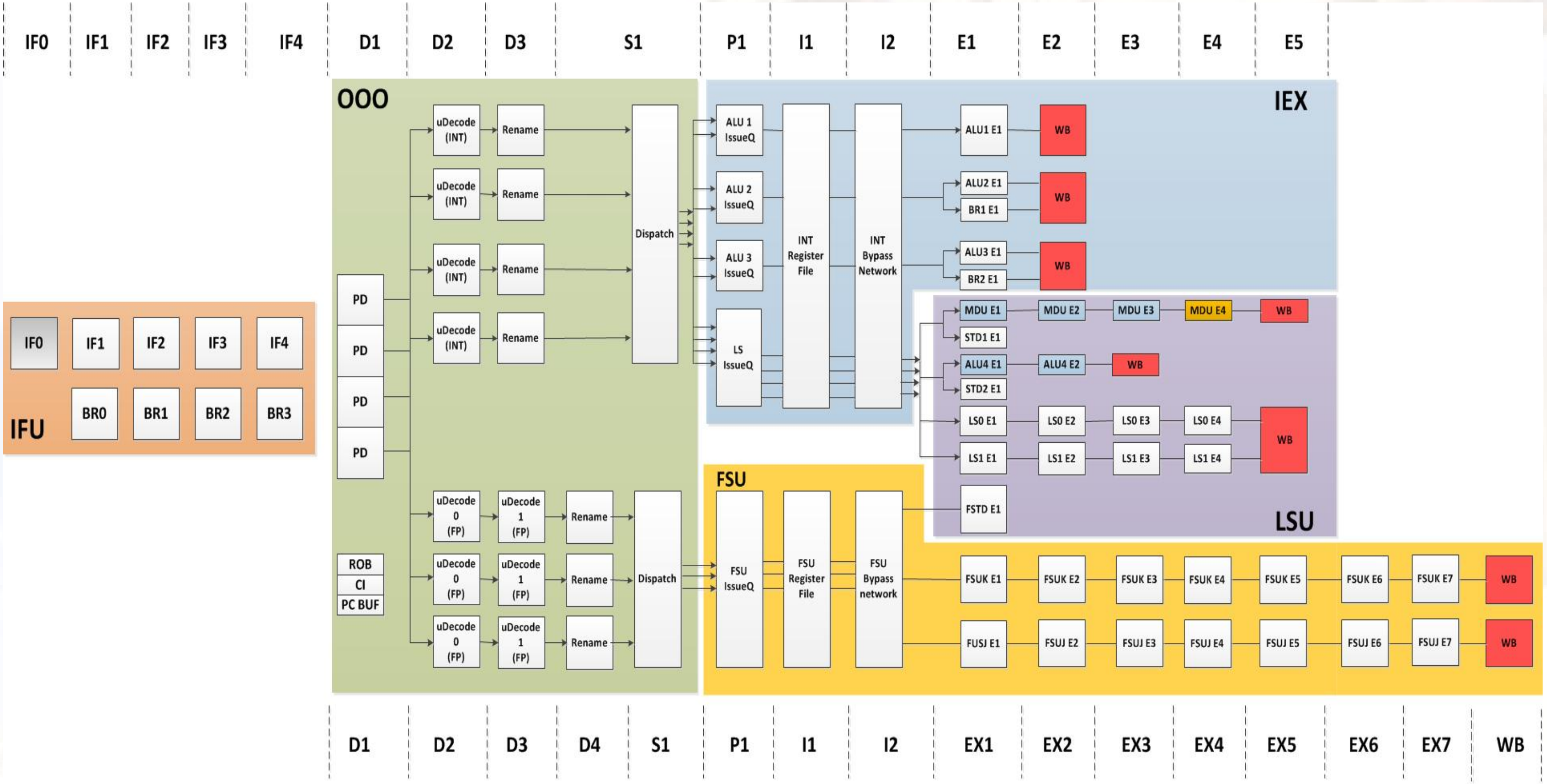
	记分牌	Tomasulo	带投机的Tomasulo
主要思想	动态调度、按序发射、乱序执行、乱序完成、集中控制。	动态调度、按序发射、乱序执行、乱序完成分散控制。增加RS（保留站）,CDB。	转移预测, 动态调度、投机执行、按序完成。在RS基础上再增加ROB(reorder buffer)。
解决何种竞争	RAW数据竞争。 WAW,WAR时stall	解决数据竞争,避免WW,WAR竞争;允许硬件支持的循环展开，不局限于程序基本块。	解决数据竞争,避免WW,WAR竞争;从数据竞争扩充到控制竞争。通过预测减少控制竞争stall。
指令节拍	ISSUE：停顿于结构竞争和WAW竞争。 RO：等待两个操作数就绪,动态解决RAW。 EX：和普通流水线一样。 WB：停顿于WAR竞争。	ISSUE：仅停顿于结构竞争（RS部件）。 EX：利用保留站（RS）进行换名，解决了WAW和WAR问题（即名字相关）。 WB：回写。	ISSUE：仅停顿于结构竞争（RS部件和ROB部件）。 EX：等待操作数就绪，执行。 WB：结果写到ROB，其它指令可以引用该临时结果，但不更新寄存器。 COMMIT：保证按序完成，保证精确中断。
关键部件	记分牌、集中控制、指令状态、功能部件状态、寄存器结果状态。	CDB(通用数据总线)：执行结果通过CDB传递，及时。 RS(保留站)：换名功能以及暂存临时数据。	ROB(REORDER BUFFER)重构序缓冲器：对移到顶部的指令进行判断。。。ROB负责寄存器重命名、暂存执行后和交付前的结果、
其它	从静态调度到动态调度的过渡算法。	一旦指令将临时结果放在保留站，则其它后续的指令都能立即在寄存器组中看见这个结果。	某指令将临时结果放在ROB后，其它指令并不能从寄存器组中看到这个结果（也就是说，寄存器组的值并没有更新），而是等待该指令提交（commit）后，寄存器组才更新。

ARM处理器体系结构

❑ARM流水线的执行顺序：

- 取指令（Fetch）：从存储器读取指令；
- 译码（Decode）：译码以鉴别它是属于哪一条指令；
- 执行（Execute）：将操作数进行组合以得到结果或存储器地址；
- 缓冲/数据（Buffer/data）：如果需要，则访问存储器以存储数据；
- 回写：（Write-back）：将结果写回到寄存器组中；

基于ARMv8的鲲鹏流水线技术



IF0 IF1 IF2 IF3 IF4 D1 D2 D3 S1 P1 I1 I2 E1 E2 E3 E4 E5

IFU
IF0 IF1 IF2 IF3 IF4
BR0 BR1 BR2 BR3

OOO
PD PD PD PD
ROB
CI
PC BUF
uDecode (INT) → Rename
uDecode (INT) → Rename
uDecode (INT) → Rename
uDecode (INT) → Rename
Dispatch
uDecode 0 (FP) → uDecode 1 (FP) → Rename
uDecode 0 (FP) → uDecode 1 (FP) → Rename
uDecode 0 (FP) → uDecode 1 (FP) → Rename
Dispatch

IEX
ALU 1 IssueQ
ALU 2 IssueQ
ALU 3 IssueQ
LS IssueQ
INT Register File
INT Bypass Network
ALU1 E1 → WB
ALU2 E1, BR1 E1 → WB
ALU3 E1, BR2 E1 → WB

LSU
MDU E1 → MDU E2 → MDU E3 → MDU E4 → WB
STD1 E1
ALU4 E1 → ALU4 E2 → WB
STD2 E1
LS0 E1 → LS0 E2 → LS0 E3 → LS0 E4
LS1 E1 → LS1 E2 → LS1 E3 → LS1 E4
WB
FSTD E1

FSU
FSU IssueQ
FSU Register File
FSU Bypass network
FSUK E1 → FSUK E2 → FSUK E3 → FSUK E4 → FSUK E5 → FSUK E6 → FSUK E7 → WB
FSUJ E1 → FSUJ E2 → FSUJ E3 → FSUJ E4 → FSUJ E5 → FSUJ E6 → FSUJ E7 → WB

D1 D2 D3 D4 S1 P1 I1 I2 EX1 EX2 EX3 EX4 EX5 EX6 EX7 WB

Taishan coreV110 Pipeline Architecture

基于ARMv8的鲲鹏流水线技术

- Branch预测和取指流水线解耦设计，取指流水线每拍最多可提供32Bytes指令供译码，分支预测流水线可以不受取指流水停顿影响，超前进行预测处理；
- 定浮点流水线分开设计，解除定浮点相互反压，每拍可为后端执行部件提供4条整型微指令及3条浮点微指令；
- 整型运算单元支持每拍4条ALU运算（含2条跳转）及1条乘除运算；
- 浮点及SIMD运算单元支持每拍2条ARM Neon 128bits 浮点及SIMD运算；
- 访存单元支持每拍2条读或写访存操作，读操作最快4拍完成，每拍访存带宽为2x128bits读及1x128bits写；