Superscalar Architecture

Tomasulo Algorithm and Memory Data Flow

Computer System Architecture

IIT Tirupati

7th April, 2020

jtt@iittp.ac.in

Register Data Flow

Goal:

To **ensure dynamic execution** while dealing effectively with the dependencies:

RAW : True
WAW : Output
WAR : Anti

Register Renaming

R1 ← R2 + R3
R1 ← R3 * R4

R5 ← R5 + R6
R6 ← R4 + R7

R1 ← R2 + R3
RR1 ← R3 * R4

R5 ← R5 + R6
RR6 ← R4 + R7

How this can done in hardware?

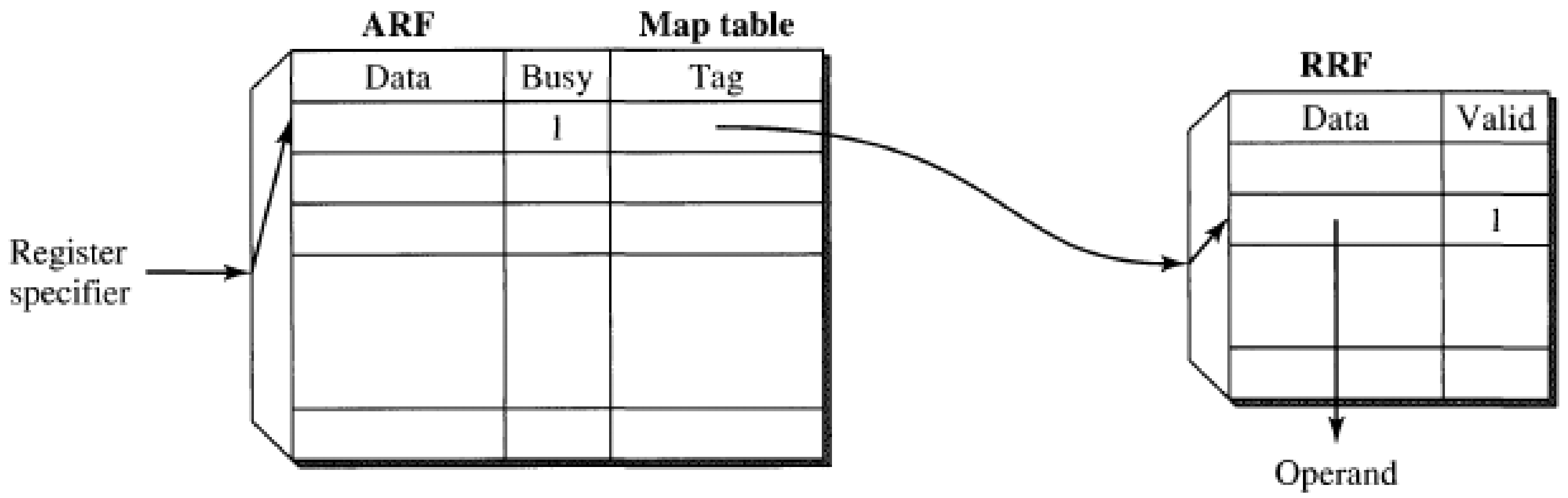
Register Renaming

R1 ← R2 + R3
R1 ← R3 * R4

R1 ← R2 + R3
RR1 ← R3 * R4

R5 ← R5 + R6
R6 ← R4 + R7

R5 ← R5 + R6
RR6 ← R4 + R7



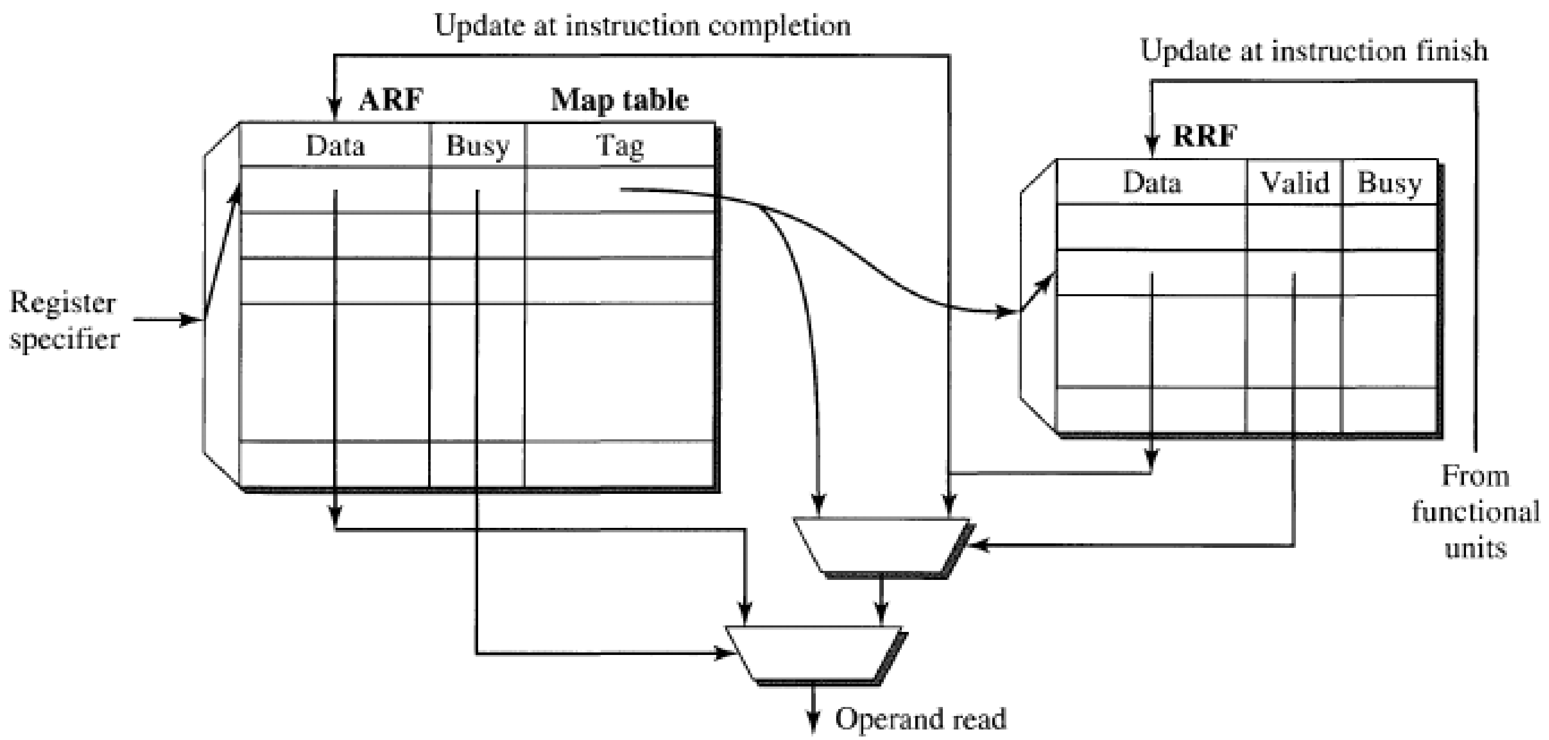
ARF – Architectural Register File
RRF – Renamed Register File

Register Renaming

R1 ← R2 + R3
R1 ← R3 * R4

Updating the value in RRF and ARF at finish and complete

R5 ← R5 + R6
R6 ← R4 + R7



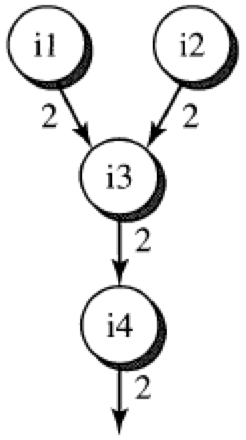
True Data Dependency

Read after Write (RAW): one of the challenge for parallel execution

```
i1:  f2 ← load,4(r2)
i2:  f0 ← load,4(r5)
i3:  f0 ← fadd,f2,f0
i4:  4(r6) ← store,f0
i5:  f14 ← laod,8(r7)
i6:  f6 ← load,0(r2)
i7:  f5 ← load,0(r3)
i8:  f5 ← fsub,f6,f5
i9:  f4 ← fmul,f14,f5
i10: f15 ← load,12(r7)
i11: f7 ← load,4(r2)
i12: f8 ← load,4(r3)
i13: f8 ← fsub,f7,f8
i14: f8 ← fmul,f15,f8
i15: f8 ← fsub,f4,f8
i16: 0(r8) ← store,f8
```

Analyse latency and *data flow limit*

Let ADD, SUB, LOAD takes 2 cycles
MUL and DIV takes 4 cycles



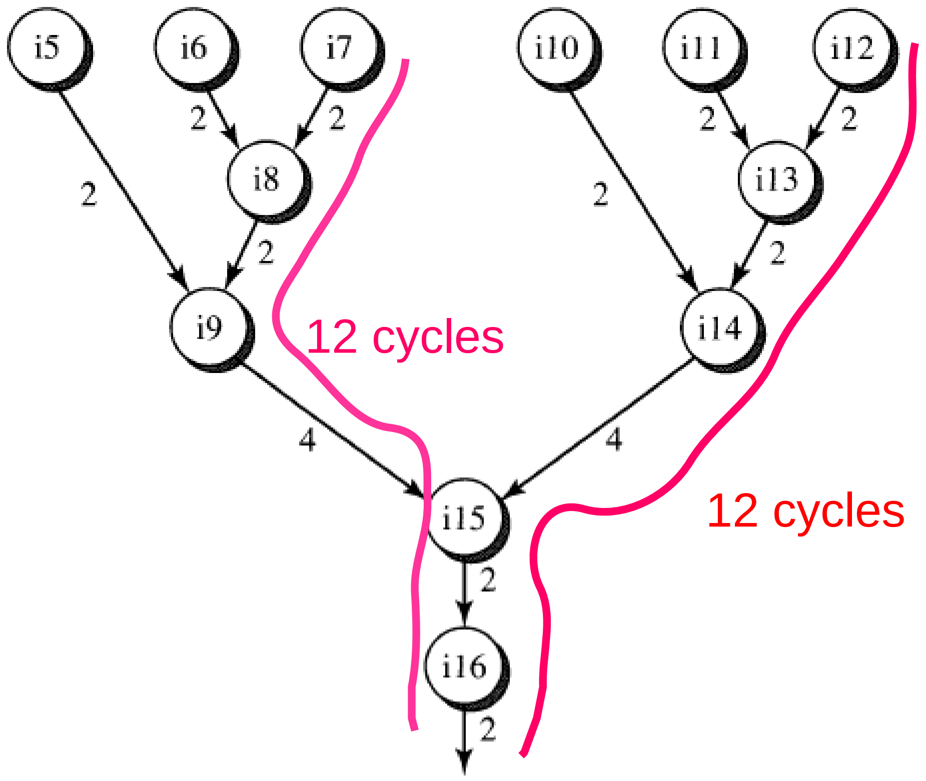
Data flow graph

True Data Dependency

Read after Write (RAW): one of the challenge for parallel execution

```
i1:  f2 ← load,4(r2)
i2:  f0 ← load,4(r5)
i3:  f0 ← fadd,f2,f0
i4:  4(r6) ← store,f0
i5:  f14 ← laod,8(r7)
i6:  f6 ← load,0(r2)
i7:  f5 ← load,0(r3)
i8:  f5 ← fsub,f6,f5
i9:  f4 ← fmul,f14,f5
i10: f15 ← load,12(r7)
i11: f7 ← load,4(r2)
i12: f8 ← load,4(r3)
i13: f8 ← fsub,f7,f8
i14: f8 ← fmul,f15,f8
i15: f8 ← fsub,f4,f8
i16: 0(r8) ← store,f8
```

Data Flow Graph (DFG)

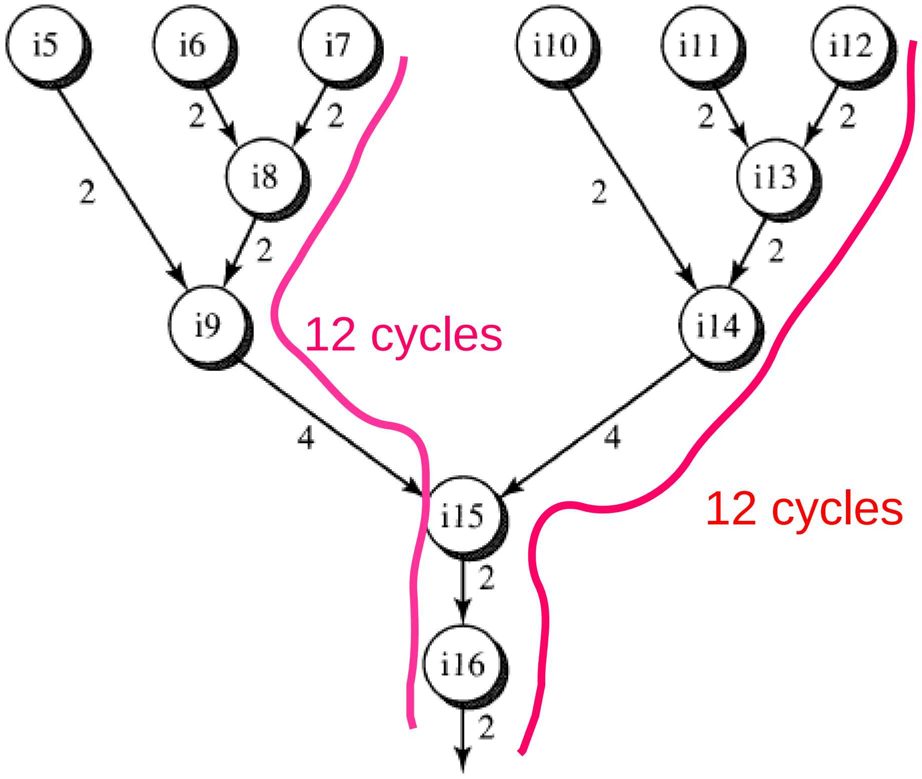


True Data Dependency

Read after Write (RAW): one of the challenge for parallel execution

```
i1:  f2 ← load,4(r2)
i2:  f0 ← load,4(r5)
i3:  f0 ← fadd,f2,f0
i4:  4(r6) ← store,f0
i5:  f14 ← laod,8(r7)
i6:  f6 ← load,0(r2)
i7:  f5 ← load,0(r3)
i8:  f5 ← fsub,f6,f5
i9:  f4 ← fmul,f14,f5
i10: f15 ← load,12(r7)
i11: f7 ← load,4(r2)
i12: f8 ← load,4(r3)
i13: f8 ← fsub,f7,f8
i14: f8 ← fmul,f15,f8
i15: f8 ← fsub,f4,f8
i16: 0(r8) ← store,f8
```

Data Flow Graph (DFG)

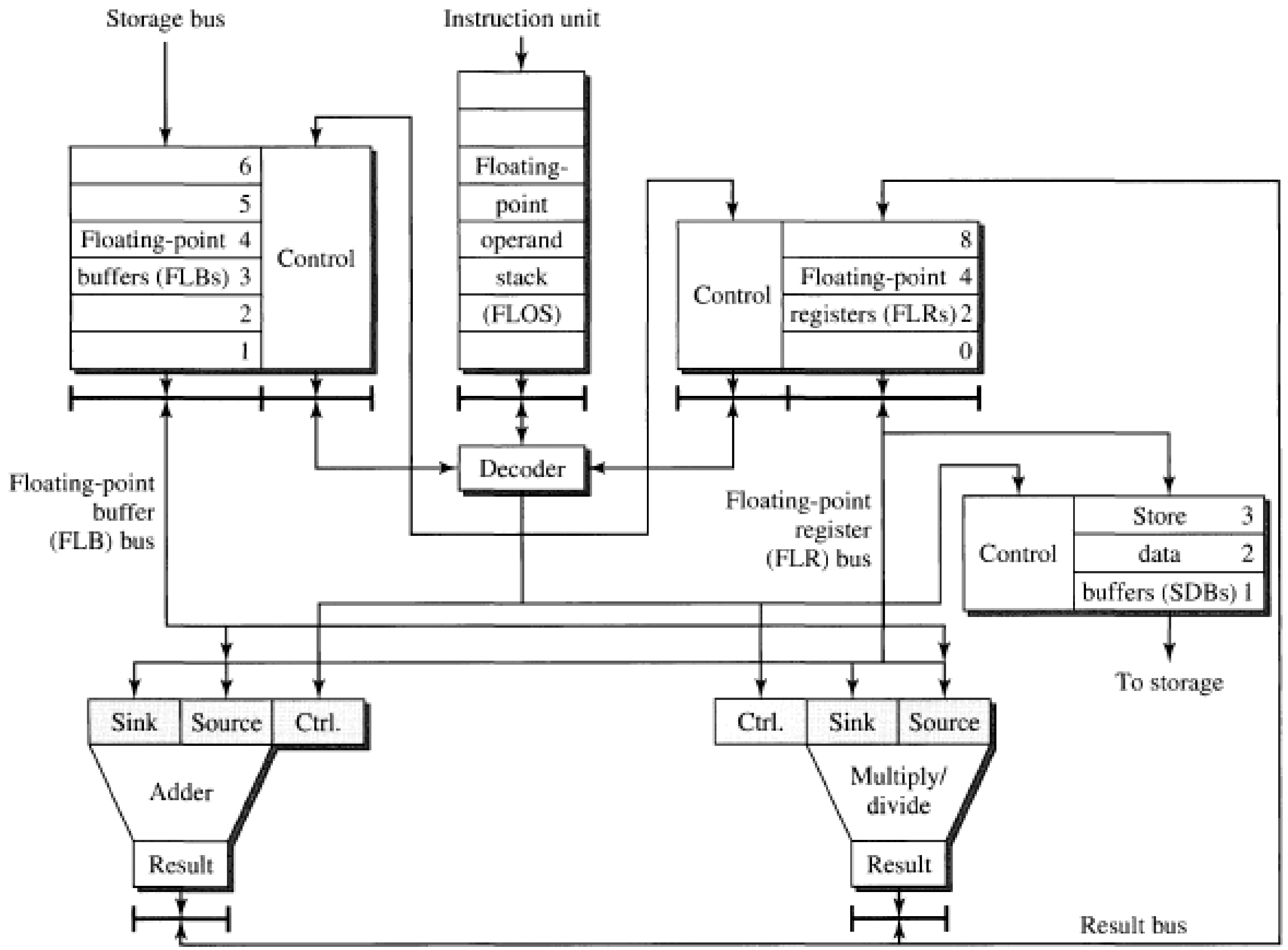


How to Ensure Data Flow

Hardware implementation for data-flow techniques:

IBM FP unit processor

(without tomasulo)



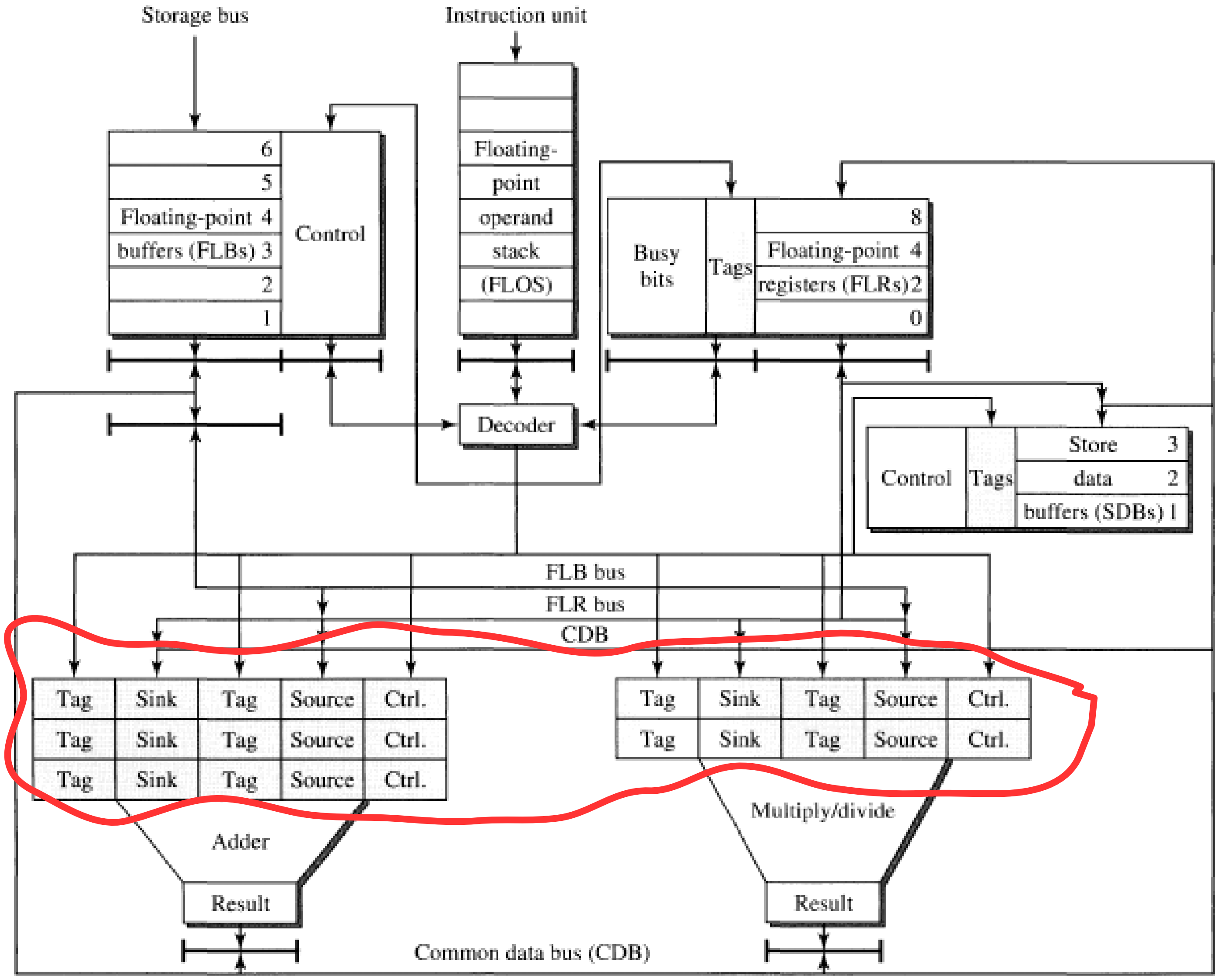
How to Ensure Data Flow

Hardware implementation for data-flow techniques:

With Tomasulo Algorithm

(IBM machine),

Almost all the cpu uses this



Tomasulo Algorithm

Working of Tomasulo's Algorithm:

RS	Tag	Sink	Tag	Source	(Reservation Station)

FLR	Busy	Tag	Data	(Floating Point Registers)

SDB	Tag	Data	(Store Data Buffer)

Tomasulo Algorithm

Working of Tomasulo's Algorithm:

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

Tomasulo Algorithm

Cycle 1: Dispatched instructions: w, x (in order)

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

RS

		Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
	2				
	3				

w Adder

RS

		Tag	Sink	Tag	Source
x	4	0	6.0	1	---
	5				

Mult/Div

FLR

	Busy	Tag	Data
0			6.0
2	yes	4	3.5
4	yes	1	10.0
8			7.8

Tomasulo Algorithm

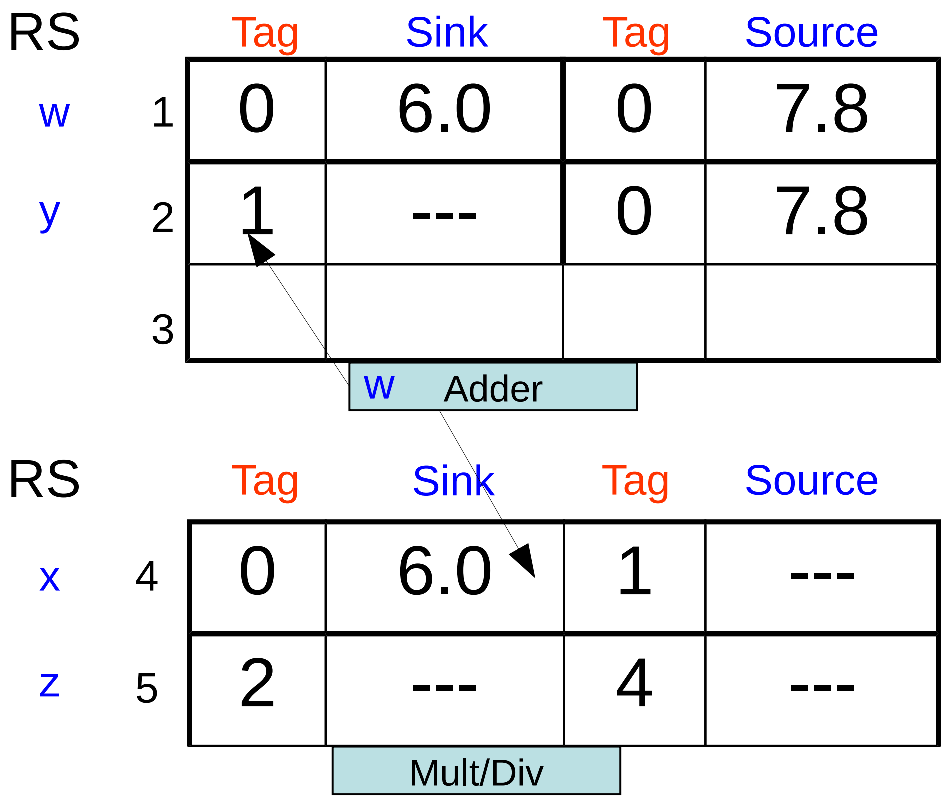
Cycle 2: Dispatched instructions: y, z (in order)

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2



RS

		Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
y	2	1	---	0	7.8
	3				

RS

		Tag	Sink	Tag	Source
x	4	0	6.0	1	---
z	5	2	---	4	---

FLR

	Busy	Tag	Data
0			6.0
2	yes	4	3.5
4	yes	2	10.0
8	yes	5	7.8

Tomasulo Algorithm

Cycle 3: Dispatched instructions: ______________

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

RS

		Tag	Sink	Tag	Source
	1				
y	2	0	13.8	0	7.8
	3				

y Adder

RS

		Tag	Sink	Tag	Source
x	4	0	6.0	0	13.8
z	5	2	---	4	---

x Mult/Div

FLR

	Busy	Tag	Data
0			6.0
2	yes	4	3.5
4	yes	2	10.0
8	yes	5	7.8

Tomasulo Algorithm

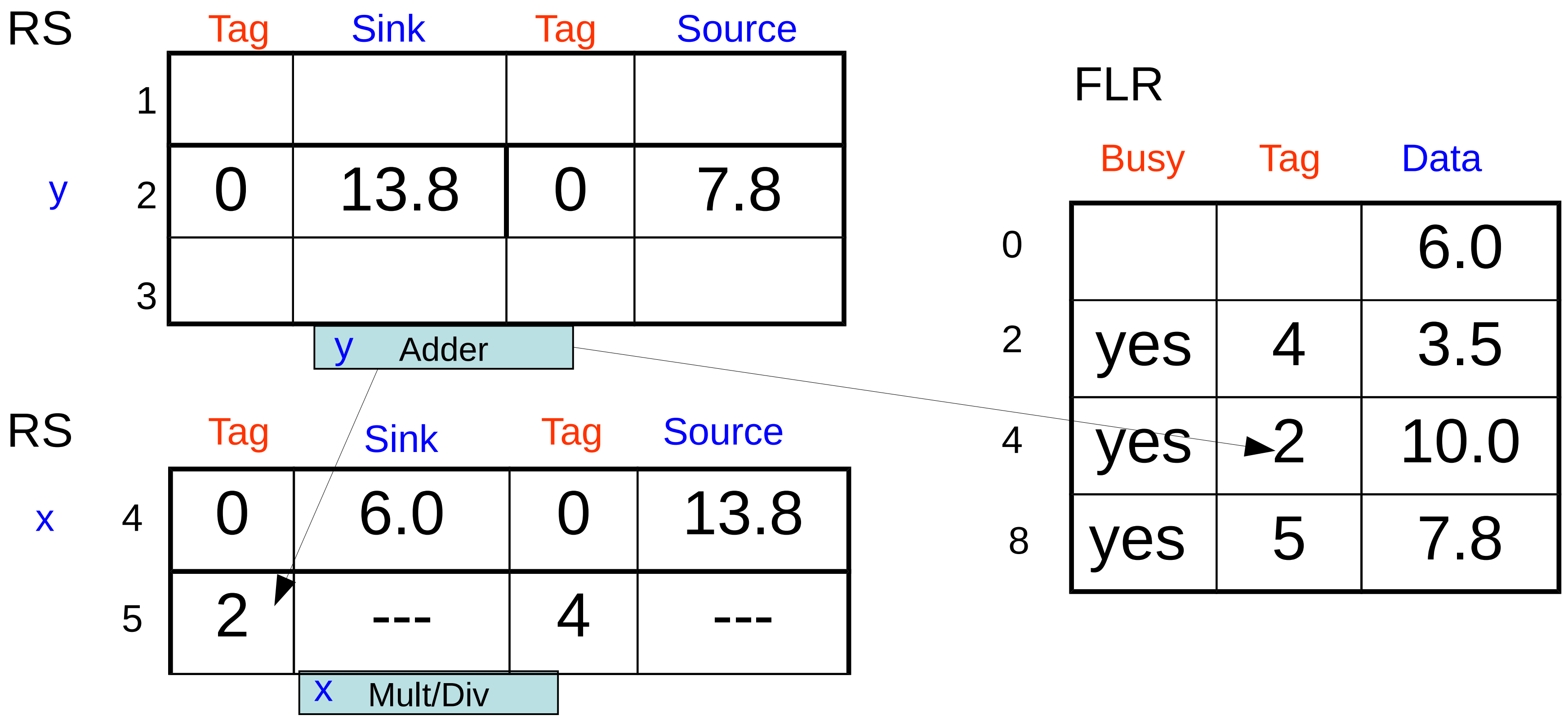
Cycle 4: Dispatched instructions:____________

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2



RS

		Tag	Sink	Tag	Source
	1				
y	2	0	13.8	0	7.8
	3				

RS

		Tag	Sink	Tag	Source
x	4	0	6.0	0	13.8
	5	2	---	4	---

FLR

	Busy	Tag	Data
0			6.0
2	yes	4	3.5
4	yes	2	10.0
8	yes	5	7.8

Tomasulo Algorithm

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

Cycle 5: Dispatched instructions:____________

RS

	Tag	Sink	Tag	Source
1				
2				
3				

Adder

RS

		Tag	Sink	Tag	Source
x	4	0	6.0	0	13.8
z	5	0	21.6	4	---

Mult/Div

FLR

	Busy	Tag	Data
0			6.0
2	yes	4	3.5
4			21.6
8	yes	5	7.8

Tomasulo Algorithm

Cycle 6: Dispatched instructions:____________

w: $R4 \leftarrow R0 + R8$

x: $R2 \leftarrow R0 * R4$

y: $R4 \leftarrow R4 + R8$

z: $R8 \leftarrow R4 * R2$

RS

	Tag	Sink	Tag	Source
1				
2				
3				

Adder

RS

		Tag	Sink	Tag	Source
	4				
z	5	0	21.6	0	82.8

(z) Mult/Div

FLR

	Busy	Tag	Data
0			6.0
2			82.8
4			21.6
8	yes	5	7.8

Tomasulo Algorithm

Cycle 7: Dispatched instructions:_____________

w: R4 ← R0 + R8
x: R2 ← R0 * R4
y: R4 ← R4 + R8
z: R8 ← R4 * R2

RS

	Tag	Sink	Tag	Source
1				
2				
3				

Adder

RS

		Tag	Sink	Tag	Source
	4				
z	5	0	21.6	0	82.8

Mult/Div

FLR

	Busy	Tag	Data
0			6.0
2			82.8
4			21.6
8	yes	5	7.8

Tomasulo Algorithm

Cycle 8: Dispatched instructions:_____________

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

RS

	Tag	Sink	Tag	Source
1				
2				
3				

Adder

RS

		Tag	Sink	Tag	Source
	4				
z	5	0	21.6	0	82.8

Mult/Div

FLR

	Busy	Tag	Data
0			6.0
2			82.8
4			21.6
8	yes	5	7.8

Tomasulo Algorithm

Cycle 9: Dispatched instructions:____________

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

RS

	Tag	Sink	Tag	Source
1				
2				
3				

Adder

RS

	Tag	Sink	Tag	Source
4				
5	0			

Mult/Div

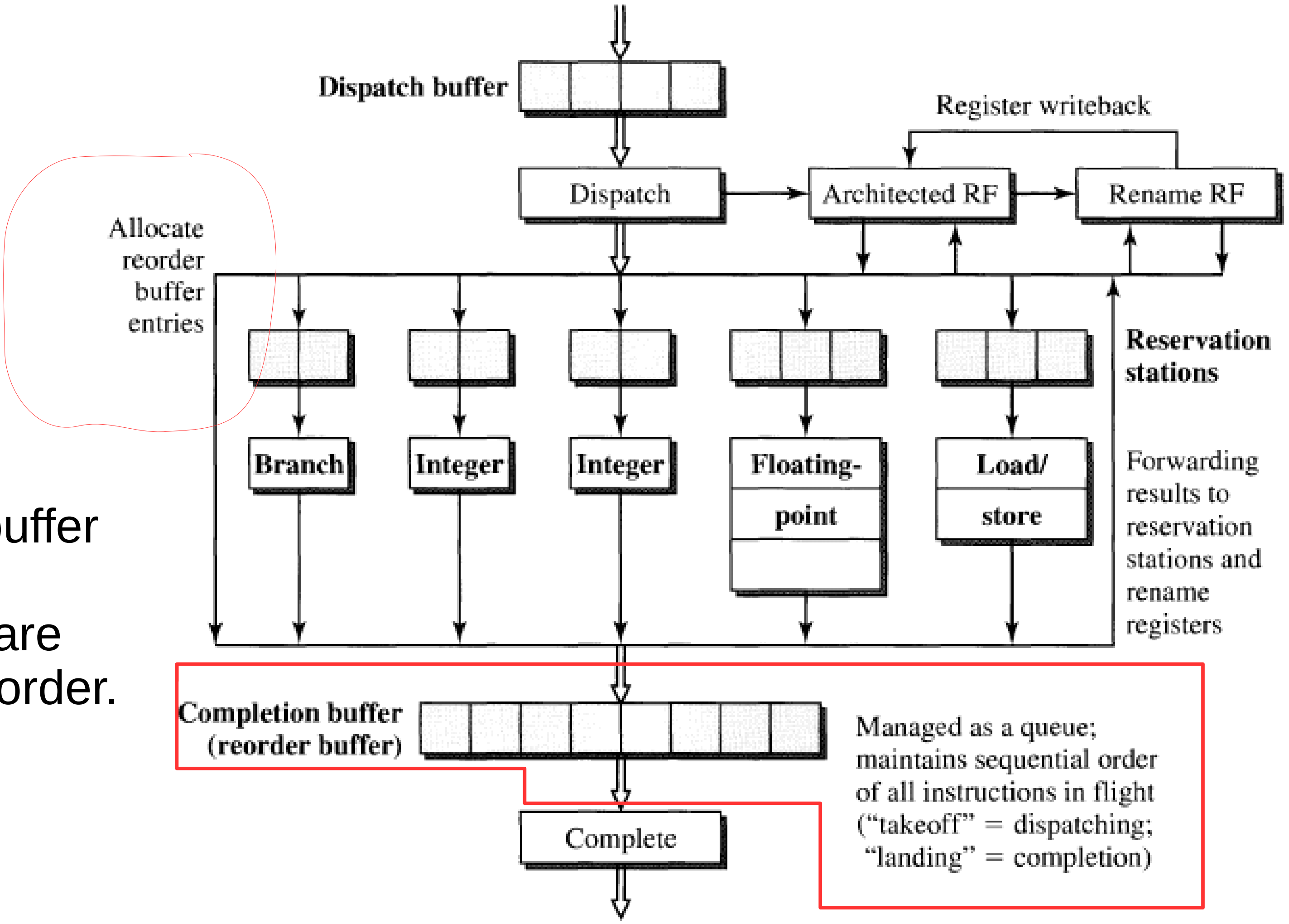
FLR

	Busy	Tag	Data
0			6.0
2			82.8
4			21.6
8			1788.48

After retirement of w, x, y, z

Dynamic Execution Core

Incorporating the Tomasulo's Algorithm in modern out-of-order core.


Dispatch buffer
Register writeback
Dispatch
Architected RF
Rename RF
Allocate reorder buffer entries
Reservation stations
Branch
Integer
Integer
Floating-point
Load/store
Forwarding results to reservation stations and rename registers
Completion buffer (reorder buffer)
Managed as a queue; maintains sequential order of all instructions in flight ("takeoff" = dispatching; "landing" = completion)
Complete

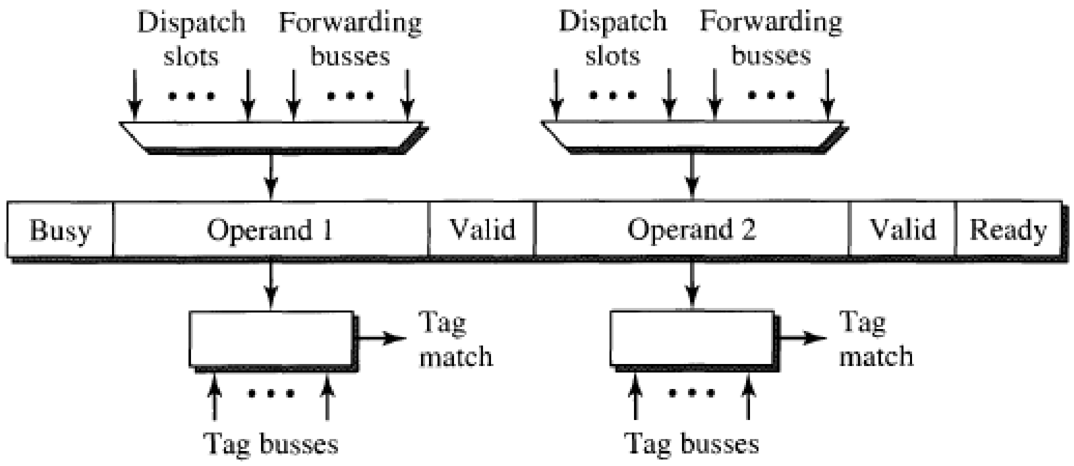
Reorder buffer gets entry when ins are issued in-order.

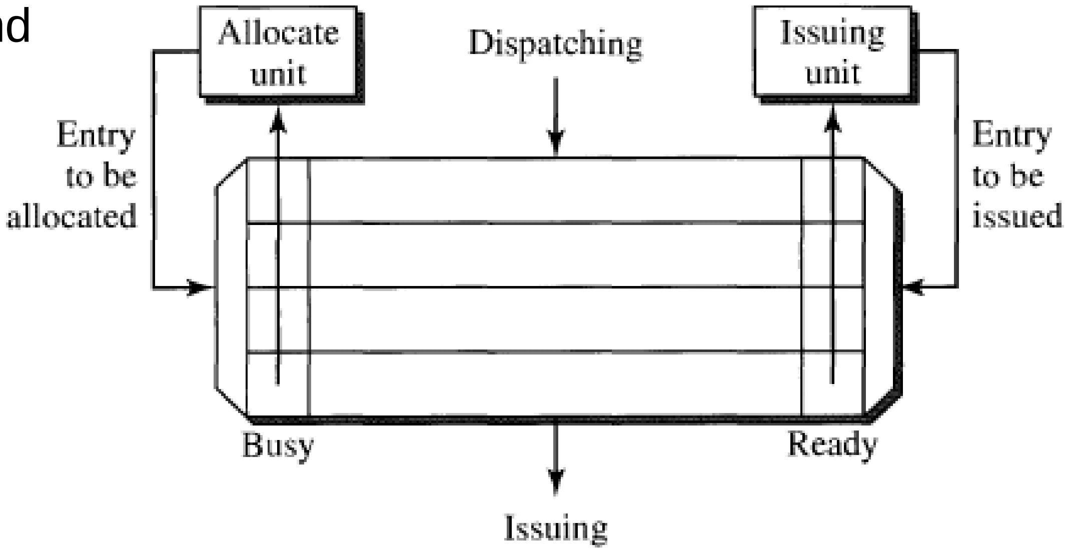
Reservation Station and ROB

Function of
Reservation Station:

- Dispatching
- Waiting
- Issuing

Busy: The entry is allocated
Ready: RS got all the source opnd
Valid: 0 for TAG in Operand
1 or actual value in Operand

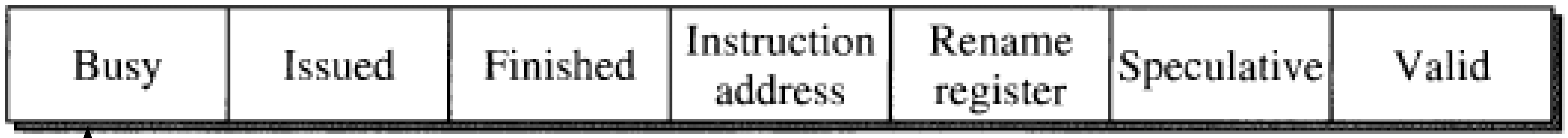




ROB

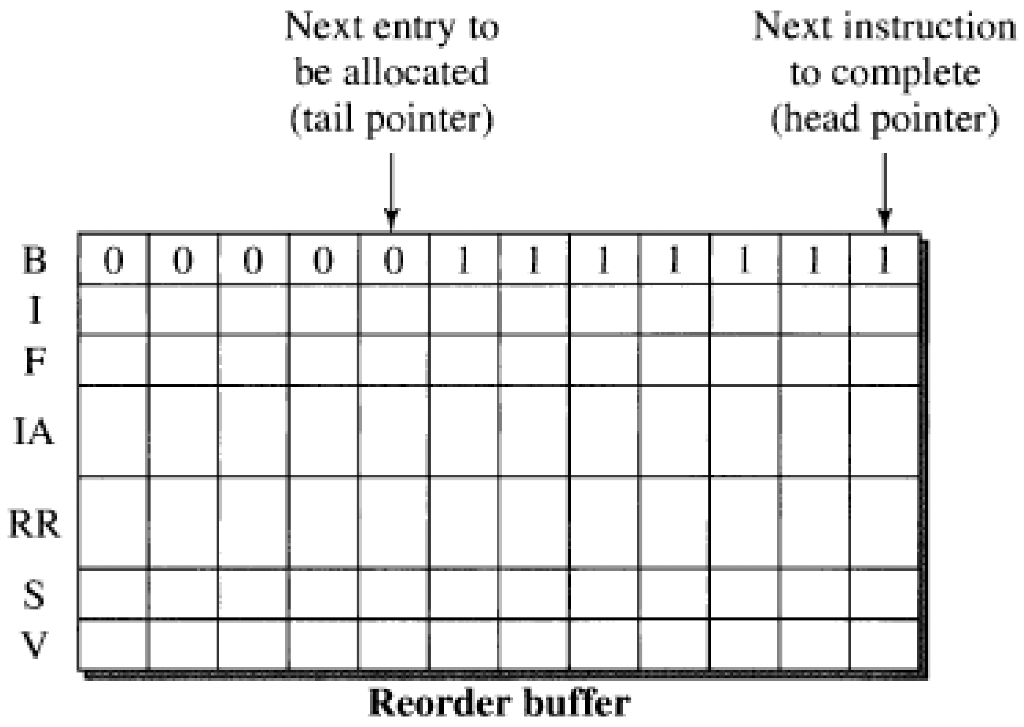
Functionality: to ensure the in-order completion of instructions

Entry updated: when the instructions are issued.



ROB Entry

ROB Organisation


Next entry to be allocated (tail pointer)
Next instruction to complete (head pointer)
B
I
F
IA
RR
S
V
0 0 0 0 0 1 1 1 1 1 1 1
Reorder buffer

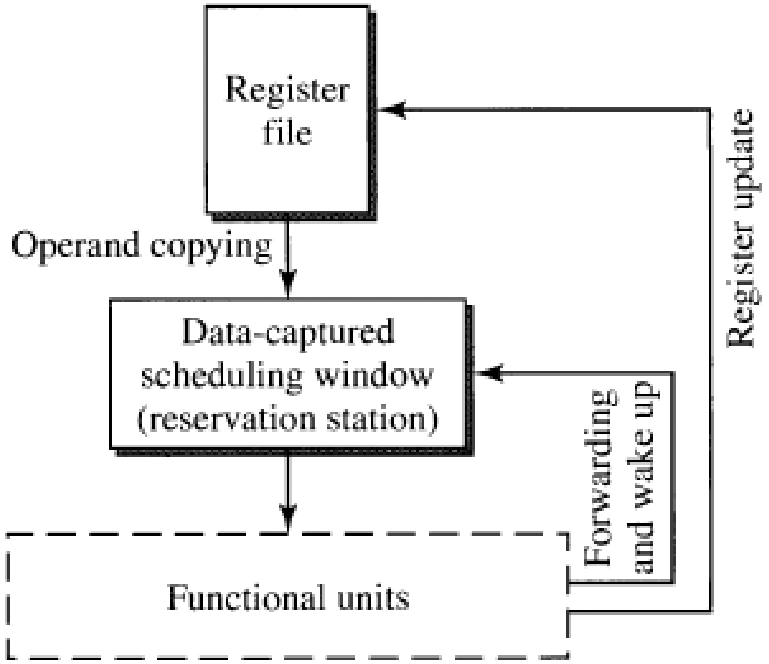
ROB contains all the Instructions which are in Reservation station and Execution.

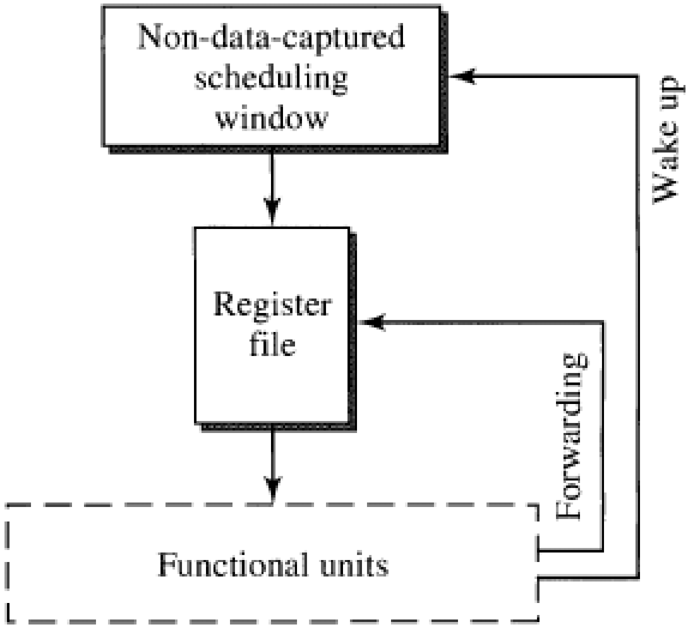
Dynamic Instruction Scheduler

WHICH instruction to be issued **WHEN** to Execution unit and **HOW**?

Read RF first and Schedule
(Data capture)

Schedule first using TAG
And then read RF
(Without Data Capture)





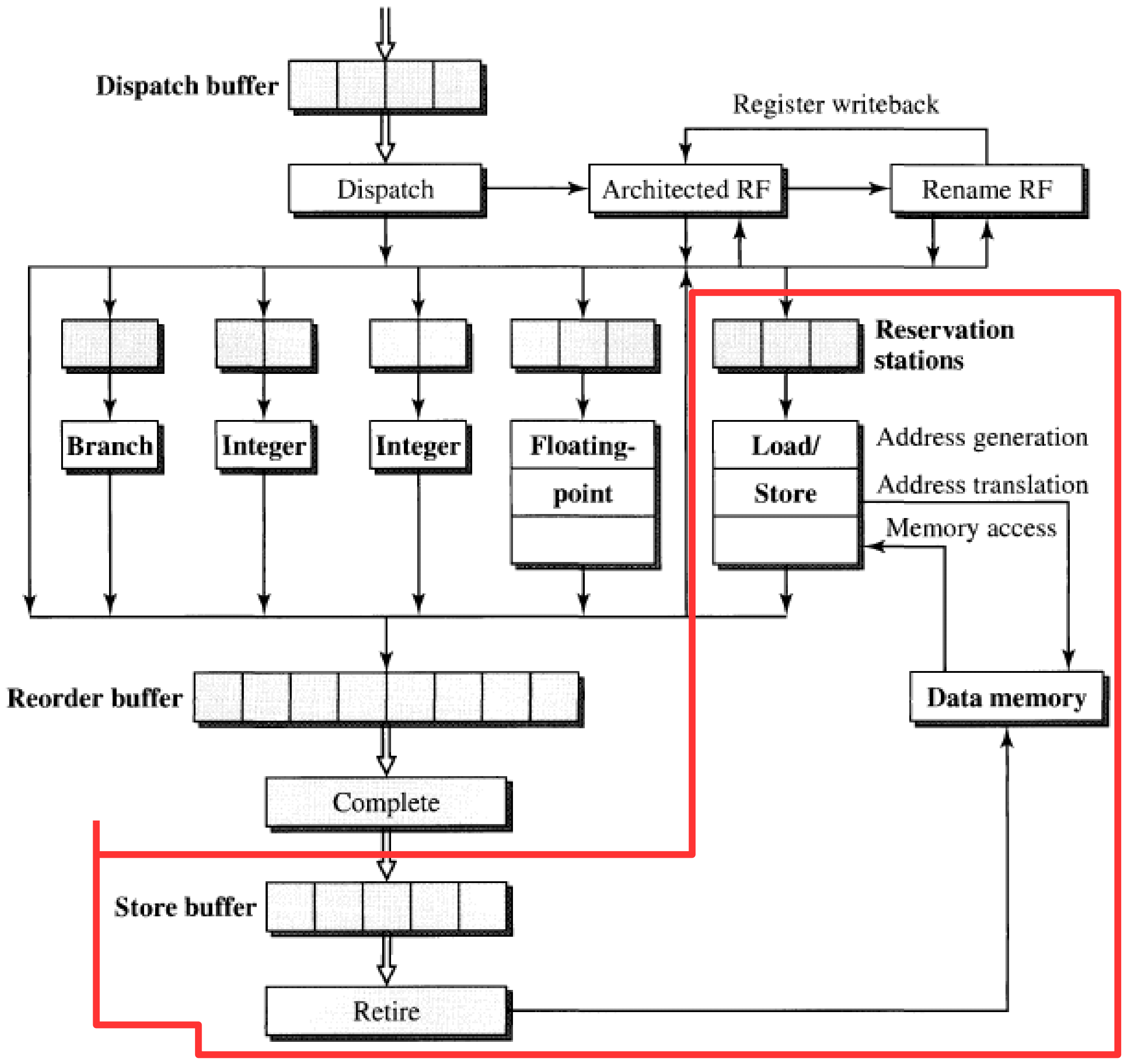
Memory Data Flow

- Limited number of registers!
- Memory is required to stored vector data structure (array, list, etc.)
- Two functions: Write (Store) and Read (Load)
  - Three steps:
    - Address generation
    - Address translation
    - Memory Access

How to deal with the depedencies among LOADs and STOREs?

Memory Data Flow

Where is the Load/Store in Execution Core?



Memory Access Ordering

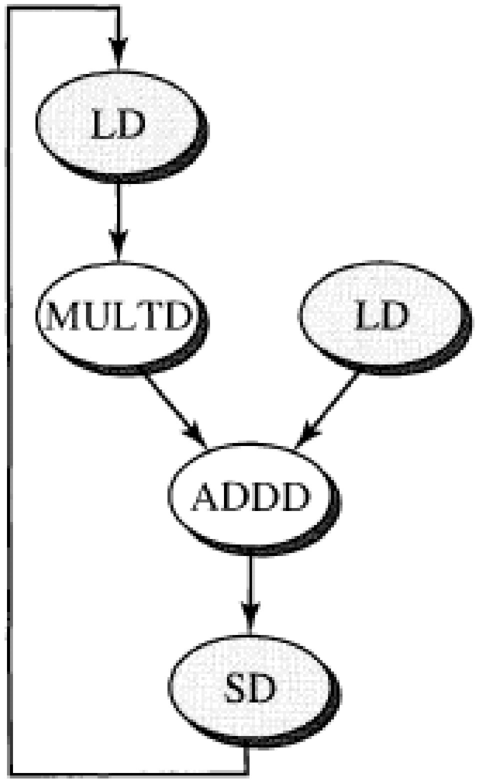
The main problem is LOAD instruction!

Dependency analysis using Dependency graph

```
  Y(i)=A*X(i)+Y(i)

     F0 ← LD,a
     R4 ← ADDI,Rx,#512       ;last address

Loop:
     F2 ← LD,0(Rx)           ;load X(i)
     F2 ← MULTD,F0,F2        ;A*X(i)
     F4 ← LD,0(Ry)           ;load Y(i)
     F4 ← ADDD,F2,F4         ;A*X(i)+Y(i)
     0(Ry) ← SD,F4           ;store into Y(i)
     Rx ← ADDI,Rx,#8         ;inc. index to X
     Ry ← ADDI,Ry,#8         ;inc. index to Y
     R20 ← SUB,R4,Rx         ;compute bound
     BNZ,R20,Loop            ;check if done
```



Memory Access Ordering

How to get the destination value of LOAD in smarter way?

Situation 1

```
ST R3, PC(index)

ST R4, DIRECT

LD base(index)  R1
```

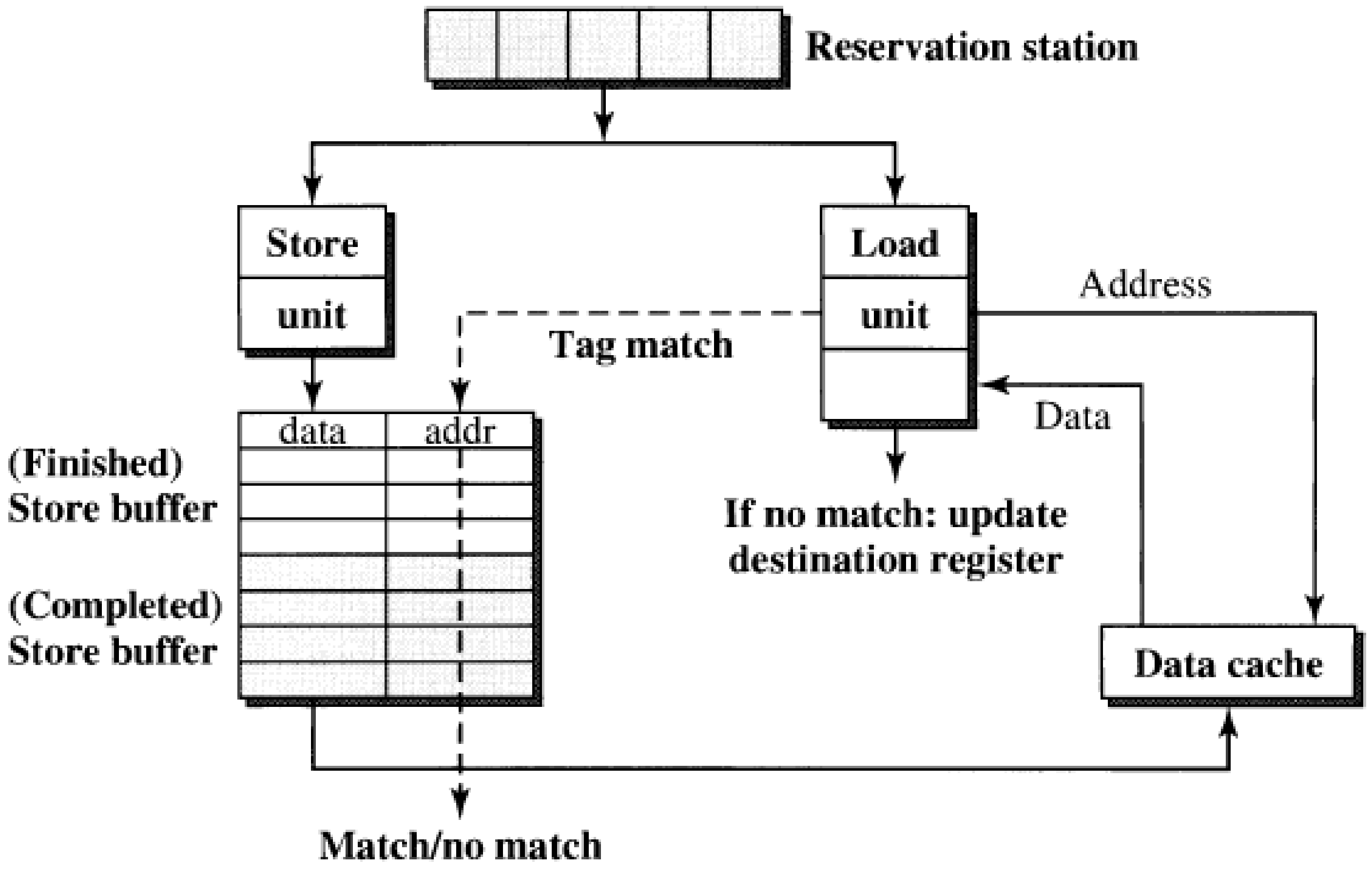
Situation 2

```
ST R3, base(index)

LD base(index)  R1
```

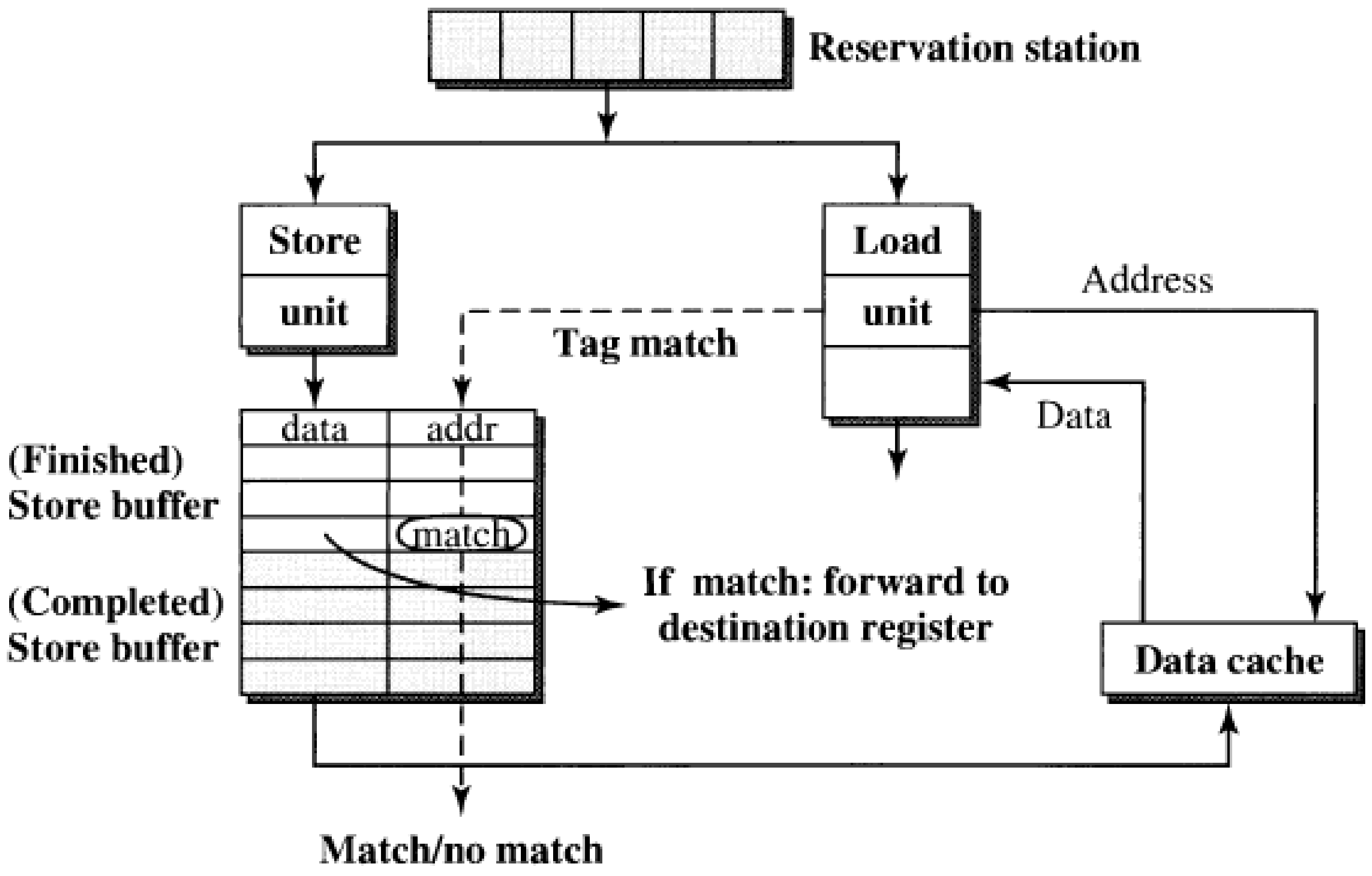
Load Bypassing

Execute the LOAD a bit early!



Load Forwarding

Let the dependence value be directly forwarded to destination of LOAD!



References

References to Dyamic super-scalar architecture

1) Chapter 4 and 5, Modern Super-scalar Processor Design, Shen and Lipasti
2) Chapter 3, Computer Architecture: Quantitative Approach
3) TAGGED Branch predictor paper

Exercises:

Exercise 4 and 5 of Shen and Lipasti
Examples from Chapter 3 of Quantitative Approach

Next Lecture

Multi-threading
Multi-core