Single-Issue Processor (AKA Scalar Processor) | CPI ≥ 1 - One At Best | IPC ≤ 1 - One At best

1 CPI < 1? How?

From Single-Issue Processors (AKA Scalar Processors) To:

→ **Multiple issue processors:**

- **VLIW (Very Long Instruction Word)** — ISA Support/ New ISA Needed
- **Superscalar processors** — No ISA Support or New ISA Needed

2 What if dynamic branch prediction is wrong?

→ **Speculative Tomasulo Processor** — Speculative Execution

Or delay start of execution of instructions following a branch until after the branch is resolved

Why? In case of a branch misprediction (wrong prediction), such instructions with delayed start of execution did not update registers or memory and thus easy to cancel.

Evolution of Microprocessor Performance

So far we examined static & dynamic techniques to improve the performance of single-issue (scalar) pipelined CPU designs including: static & dynamic scheduling, static & dynamic branch predication. Even with these improvements, the restriction of issuing a single instruction per cycle still limits the ideal CPI = 1

$T = I \times CPI \times C$

Multi-cycle (Not Pipelined) → Pipelined (single issue) ← Multiple Issue (CPI <1) Superscalar/VLIW/SMT

4-Bit Intel 4004	1970-1980	1980-1990	1990-2000	2000-2010
Transistor Count	2K-100K	100K-1M	1M-100M	100M-1B
Clock Frequency	0.1-3MHz	3-30MHz	30M-1GHz	1 GHz to ???? GHz
IPC: Instruction/Cycle	< 0.1	0.1-0.9	0.9- 1.9	1.9-2.9 (?)
CPI	> 10	1.1-10	0.5 - 1.1	.35 - .5 (?)

Original (2002) Intel Predictions 15 GHz

Source: John P. Chen, Intel Labs

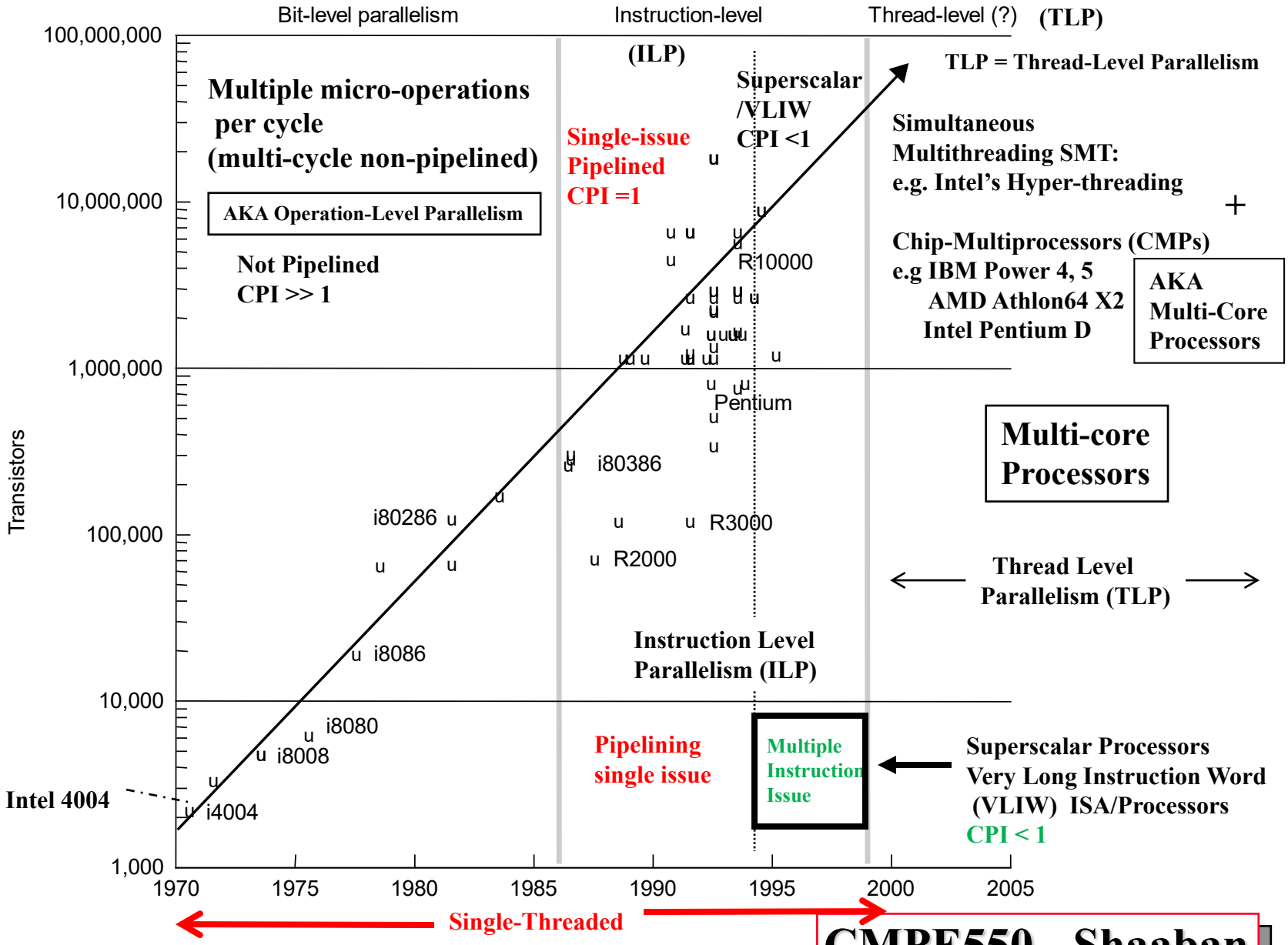
We next examine the two approaches to achieve a CPI < 1 by issuing multiple instructions per cycle:

- Superscalar CPUs
- Very Long Instruction Word (VLIW) CPUs.

4th Edition: Chapter 2.6-2.8
(3rd Edition: Chapter 3.6, 3.7, 4.3

Single-issue Processor = Scalar Processor
Instructions Per Cycle (IPC) = 1/CPI

Parallelism in Microprocessor VLSI Generations



Multiple Instruction Issue: CPI < 1

- To improve a pipeline's CPI to be better [less] than one, and to better exploit Instruction Level Parallelism (ILP), a number of instructions have to be *issued* in the same cycle.
- Multiple instruction issue processors are of two types:

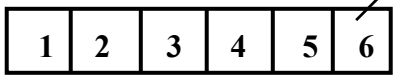
Most common = 4 instructions/cycle called 4-way superscalar processor

1 – **Superscalar:** A number of instructions (2-8) is issued in the same cycle, scheduled statically by the compiler or -more commonly- dynamically (Tomasulo).

No ISA Support or New ISA Needed

  - PowerPC, Sun UltraSparc, Alpha, HP 8000, Intel PII, III, 4 ...

2 – **VLIW (Very Long Instruction Word):** One Cycle →



A fixed number of instructions (3-6) are formatted as one long instruction word or packet (statically scheduled by the compiler).

  - Example: Explicitly Parallel Instruction Computer (EPIC)

IA-64

Special New ISA Needed

    - Originally a joint HP/Intel effort.
    - ISA: Intel Architecture-64 (IA-64) 64-bit address:
    - First CPU: Intel Itanium, Q1 2001. Itanium 2 (2003)

- **Limitations of the approaches:**
  - Available ILP in the program (both).
  - Specific hardware implementation difficulties (superscalar).
  - VLIW optimal compiler design issues.

4th Edition: Chapter 2.7
(3rd Edition: Chapter 3.6, 4.3

Simple Statically Scheduled Superscalar Pipeline

- **Two instructions can be issued per cycle** (**static two-issue or 2-way superscalar**).
- One of the instructions is integer (including load/store, branch). The other instruction is a floating-point operation.

Statically Scheduled Superscalar Example: Intel Atom Processor

  - This restriction reduces the complexity of hazard checking.
- **Hardware must fetch, decode and issue two instructions per cycle.**
- Then it determines whether zero (a stall), one or two instructions can be issued (in decode stage) per cycle.

Instruction Type	1	2	3	4	5	6	7	8
Integer Instruction	IF	ID	EX	MEM	WB			
FP Instruction	IF	ID	EX	EX	EX	WB		
Integer Instruction		IF	ID	EX	MEM	WB		
FP Instruction		IF	ID	EX	EX	EX	WB	
Integer Instruction			IF	ID	EX	MEM	WB	
FP Instruction			IF	ID	EX	EX	EX	WB
Integer Instruction				IF	ID	EX	MEM	WB
FP Instruction				IF	ID	EX	EX	EX

Two-issue statically scheduled pipeline in operation
FP instructions assumed to be adds (EX takes 3 cycles)

Instructions assumed independent (no stalls)

2-Issue: Ideal CPI = 0.5 Ideal Instructions Per Cycle (IPC) = 2

Intel IA-64: VLIW "Explicitly Parallel Instruction Computing (EPIC)"

- Three 41-bit instructions in 128 bit "Groups" or bundles; an instruction bundle template field (5-bits) determines if instructions are dependent or independent and statically specifies the functional units to used by the instructions:
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show dependencies of more than three instructions.

3-Issue

i.e statically scheduled by compiler

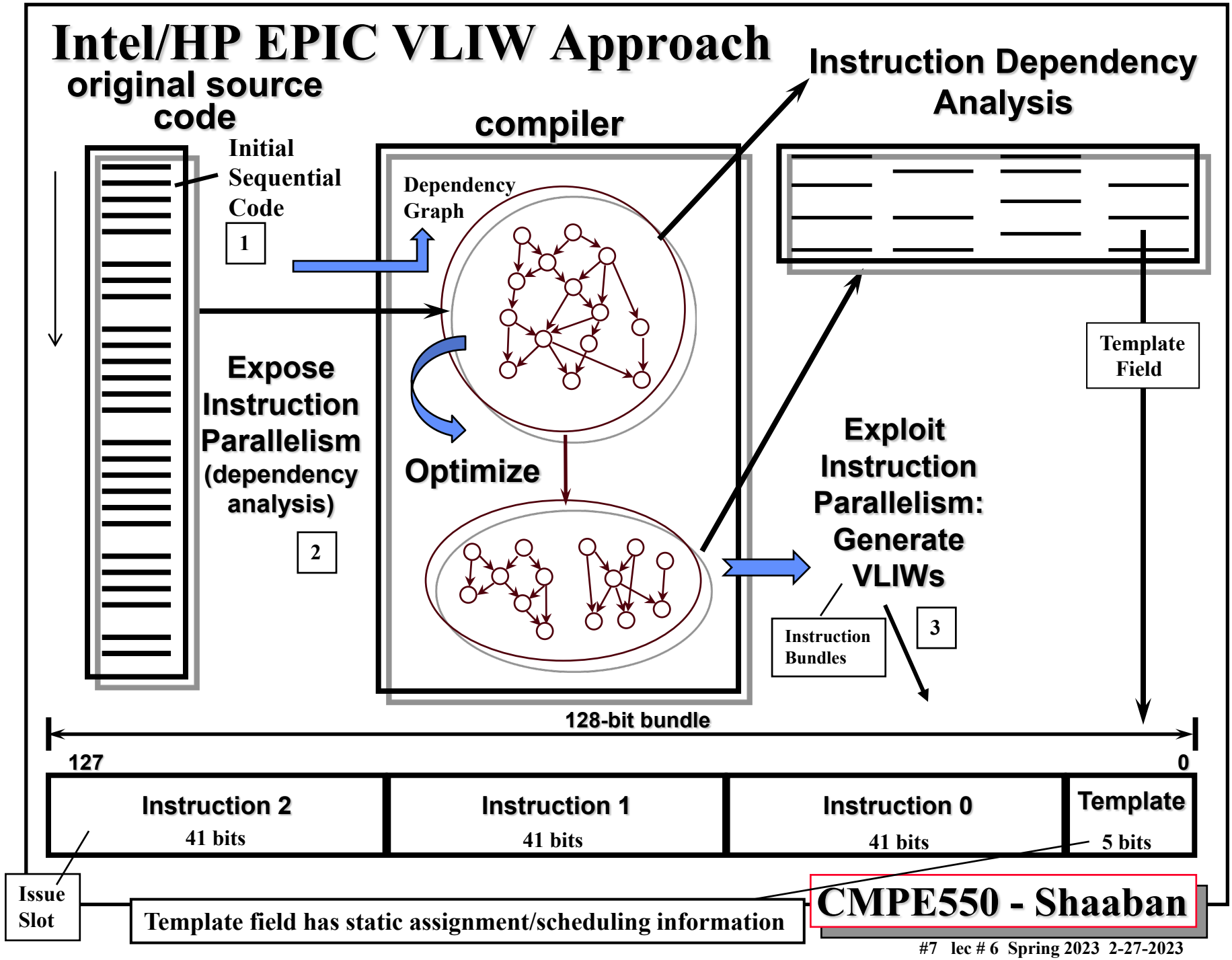
- 128 integer registers + 128 floating point registers
- Hardware checks dependencies
  (interlocks ⇒ binary compatibility over time)

Target CPUs: Statically scheduled
No register renaming in hardware

- Predicated execution: An implementation of conditional instructions used to reduce the number of conditional branches used in the generated code ⇒ larger basic block size
- IA-64 : Name given to instruction set architecture (ISA).
- Itanium : Name of the first implementation (2001).

In VLIW dependency analysis is done statically by the compiler not dynamically in hardware (Tomasulo)

Intel/HP EPIC VLIW Approach



IA-64 Instruction Types

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-integer ALU	I-unit
M	Memory	M-unit
F	Floating Point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit/B-unit

Information on static assignment of instructions to functional units and instruction typed in a bundle contained in the template field

IA-64 Template Use

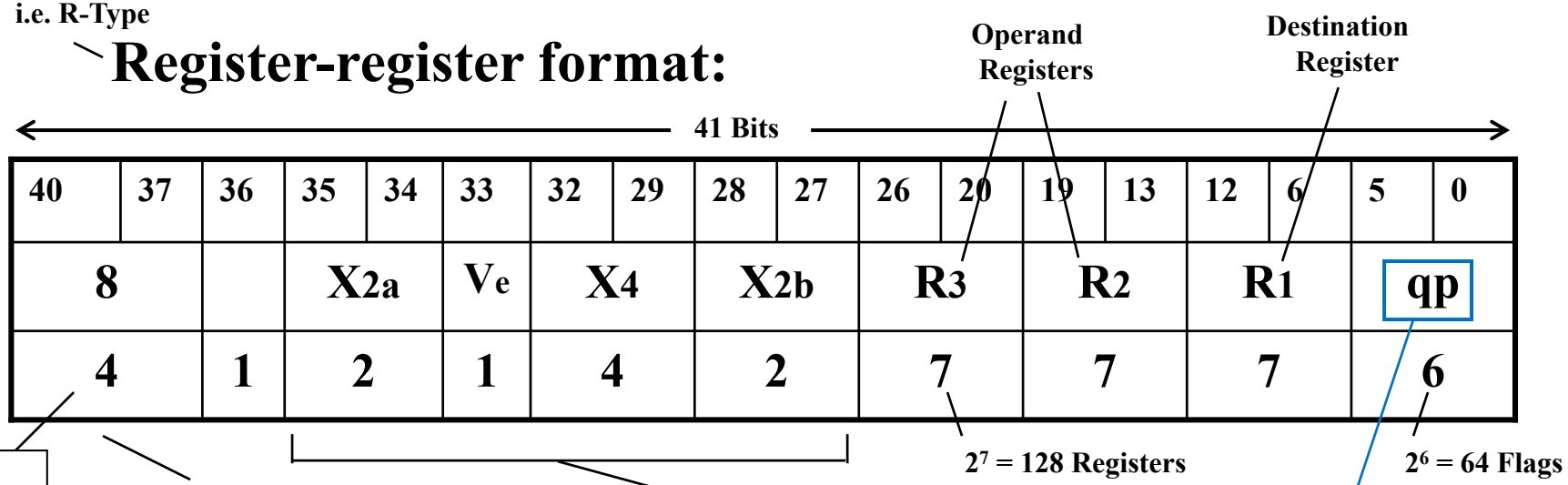
- **The 5-bit template specifies the functional units for the three operations (instructions) in the instruction bundle.**
  - **Part of static scheduling**
- **Possible instruction combinations:**
  - M-unit, I-unit, I-unit
  - M-unit, L-unit, X-unit
  - M-unit, M-unit, I-unit
  - M-unit, F-unit, I-unit
  - M-unit, M-unit, F-unit
  - M-unit, I-unit, B-unit
  - M-unit, B-unit, B-unit
  - B-unit, B-unit, B-unit
  - M-unit, M-unit, B-unit
  - M-unit, F-unit, B-unit

IA-64 Instruction Format Example:

Type A (Integer ALU) Instruction Format

i.e. R-Type

Register-register format:



- 8 is the major opcode for this instruction type.
- X2a, X2b, Ve, and X4 are opcode extensions.
- qp is the predicate register (or predication flag) assigned to this operation (64 such flags)

AKA Conditional Instruction Execution

Predication: Any instruction can be cancelled (turned into a no-op) based on the value of one of 64 predication flags (qp)

Purpose: To reduce number of branches in code (larger basic blocks)

X(i) = X(i) + S

Unrolled Loop Example for Scalar (single-issue) Pipeline

```
1 Loop: L.D     F0,0(R1)
2       L.D     F6,-8(R1)
3       L.D     F10,-16(R1)
4       L.D     F14,-24(R1)
5       ADD.D   F4,F0,F2
6       ADD.D   F8,F6,F2
7       ADD.D   F12,F10,F2
8       ADD.D   F16,F14,F2
9       S.D     F4,0(R1)
10      S.D     F8,-8(R1)
11      DADDUI  R1,R1,#-32
12      S.D     F12,16(R1)
13      BNE     R1,R2,LOOP
14      S.D     F16,8(R1)      ; 8-32 = -24
```

Latency:
L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

→ Unrolled (4-times) and scheduled loop from loop unrolling example in lecture # 3 (slide 14)

→ Recall that loop unrolling exposes more ILP by increasing size of resulting basic block

14 clock cycles, or 3.5 per original iteration (result) (unrolled four times)

3.5 = 14/4 cycles per original iteration

No stalls in code above: CPI = 1 (ignoring initial pipeline fill cycles)

Loop Unrolling in 2-way Superscalar Pipeline: (1 Integer, 1 FP/Cycle)

Unrolled 5 times

2-Issue: Ideal CPI = 0.5 IPC = 2

	1 Integer instruction	2 FP instruction	Clock cycle
Loop:	L.D F0,0(R1)		1
	L.D F6,-8(R1)		2
	L.D F10,-16(R1)	ADD.D F4,F0,F2	3
	L.D F14,-24(R1)	ADD.D F8,F6,F2	4
	L.D F18,-32(R1)	ADD.D F12,F10,F2	5
	S.D F4,0(R1)	ADD.D F16,F14,F2	6
	S.D F8,-8(R1)	ADD.D F20,F18,F2	7
	S.D F12,-16(R1)		8
	DADDUI R1,R1,#-40		9
	S.D F16,-24(R1)		10
	BNE R1,R2,LOOP		11
	SD -32(R1),F20		12

Empty or wasted issue slot

Total =7

12/5 = 2.4 cycles per original iteration

- Unrolled 5 times to avoid delays and expose more ILP (unrolled one more time)
- 12 cycles, or 12/5 = 2.4 cycles per iteration (3.5/2.4= 1.5X faster than scalar)
- CPI = 12/ 17 = .7 worse than ideal CPI = .5 because 7 issue slots are wasted

Recall that loop unrolling exposes more ILP by increasing basic block size

Scalar Processor = Single-issue Processor

Superscalar/VLIW Architecture Limitations:

i.e. Multiple-issue processors

Issue Slot Waste Classification

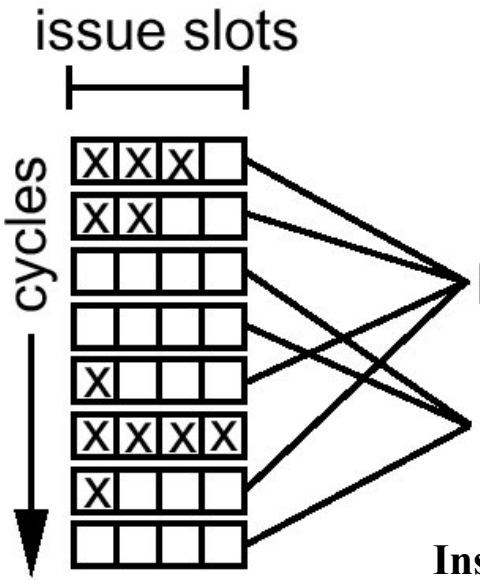
- **Empty or wasted issue slots can be defined as either vertical waste or horizontal waste:**
  - **Vertical waste is introduced when the processor issues no instructions in a cycle.** Full Stall?
  - **Horizontal waste occurs when not all issue slots can be filled in a cycle.** Partial Stall?

Example:

4-Issue Superscalar (Or VLIW)

Ideal IPC =4
Ideal CPI = ¼=0.25

Time



full issue slot

empty issue slot

horizontal waste = 9 slots (Partial Stall?)

vertical waste = 12 slots (Full Stall?)

Issue

3 x 4 =12

Also applies to VLIW

Instructions Per Cycle = IPC = 1/CPI

Or Ideal

Result of issue slot waste: Actual Performance << Peak Performance

Loop Unrolling in VLIW Pipeline (2 Memory, 2 FP, 1 Integer / Cycle)

5-issue VLIW
Ideal CPI = 0.2
IPC = 5

1 *Memory reference 1*	2 *Memory reference 2*	3 *FP operation 1*	4 *FP op. 2*	5 *Int. op/ branch*	*Clock*
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D F4,0(R1)	S.D F8, -8(R1)	ADD.D F28,F26,F2			6
S.D F12, -16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56	7
S.D F20, 24(R1)	S.D F24,16(R1)				8
S.D F28, 8(R1)				BNE R1,R2,LOOP	9

Empty or wasted issue slot

Total =22

Unrolled 7 times to avoid delays and expose more ILP

7 results in 9 cycles, or 1.3 cycles per iteration

9/7 = 1.3 cycles per original iteration

(2.4/1.3 =1.8X faster than 2-issue superscalar, 3.5/1.3 = 2.7X faster than scalar)

Average: about 23/9 = 2.55 IPC (instructions per cycle) vs. Ideal IPC =5,
CPI = 0.39 Ideal CPI = 0.2 Thus about 50% efficiency, 22 issue slots are wasted

Note: Needs more registers in VLIW (15 vs. 6 in Superscalar)

Scalar Processor = Single-Issue Processor

4th Edition: Chapter 2.7 pages 116-117
(3rd Edition: Chapter 4.3 pages 317-318)

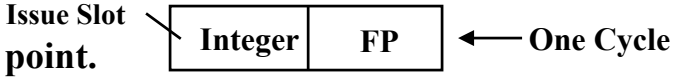
Superscalar Tomasulo-based Dynamic Scheduling

i.e. Multiple-Issue Processor

- The Tomasulo dynamic scheduling algorithm is extended to issue more than one instruction per cycle.
- However the restriction that instructions must issue in program order still holds to avoid violating instruction dependencies (to construct correct dependency graph dynamically).

The result of issuing multiple instructions in one cycle should be the same as if they were single-issued, one instruction per cycle.

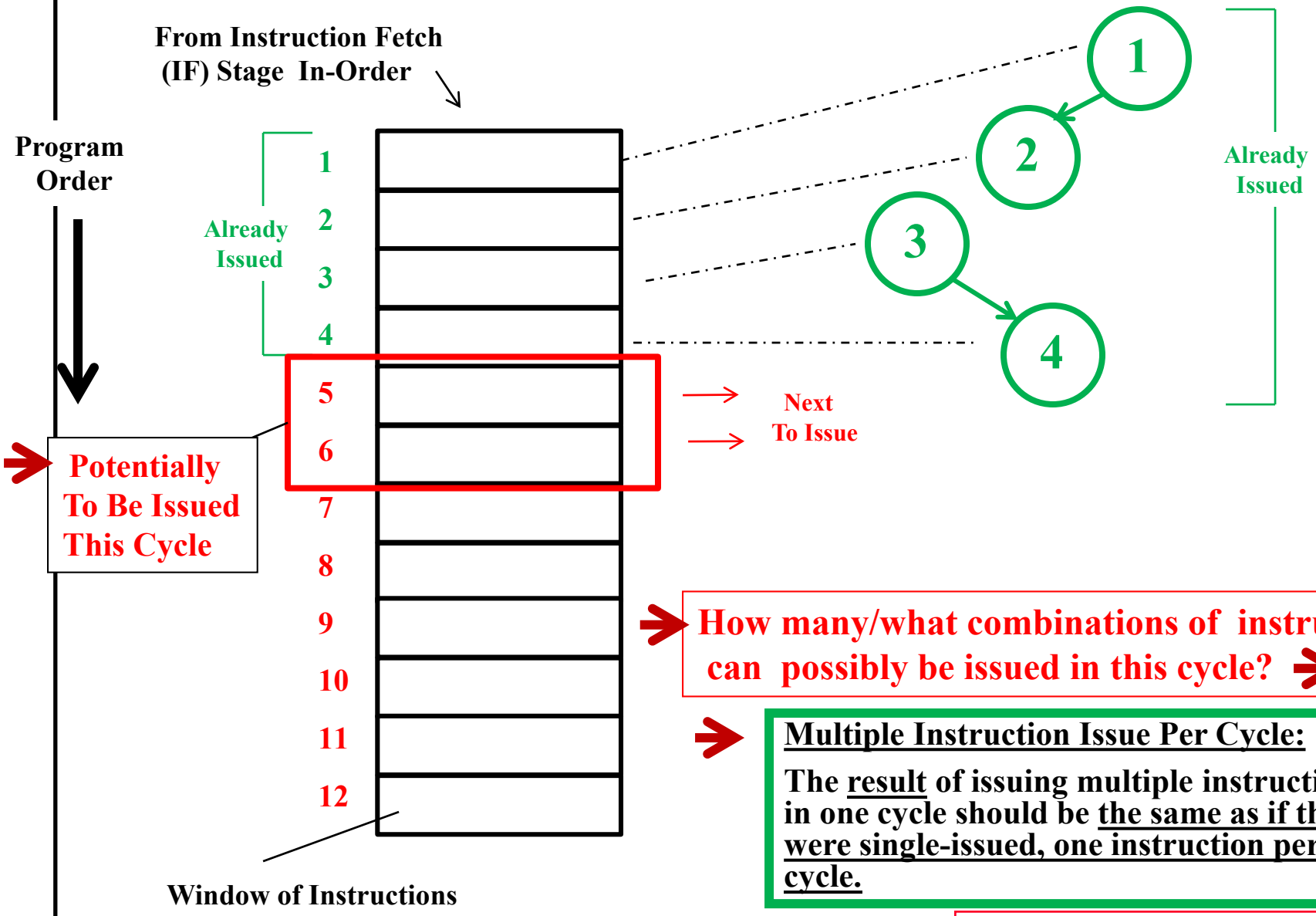
- How to issue two instructions and keep in-order instruction issue for Tomasulo?
- Simplest Method: Restrict Type of Instructions Issued Per Cycle
- To simplify the issue logic, issue one integer + one floating-point instruction per cycle (for a 2-way superscalar).
  - 1 Tomasulo control for integer, 1 for floating point.



- FP loads/stores might cause a dependency between integer and FP issue:
  - Replace load reservation stations with a load queue; operands must be read in the order they are fetched (program order).
  - Replace store reservation stations with a store queue; operands must be written in the order they are fetched.
    - Load checks addresses in Store Queue to avoid RAW violation
      - (get load value from store queue if memory address matches)
    - Store checks addresses in Load Queue to avoid WAR, and checks Store Queue to avoid WAW.

(the above load/store queue checking is also applicable to single-issue Tomasulo to take care of memory RAW, WAR, WAW). More on this later ..

Two Instruction Issue Per Cycle (In-Order)

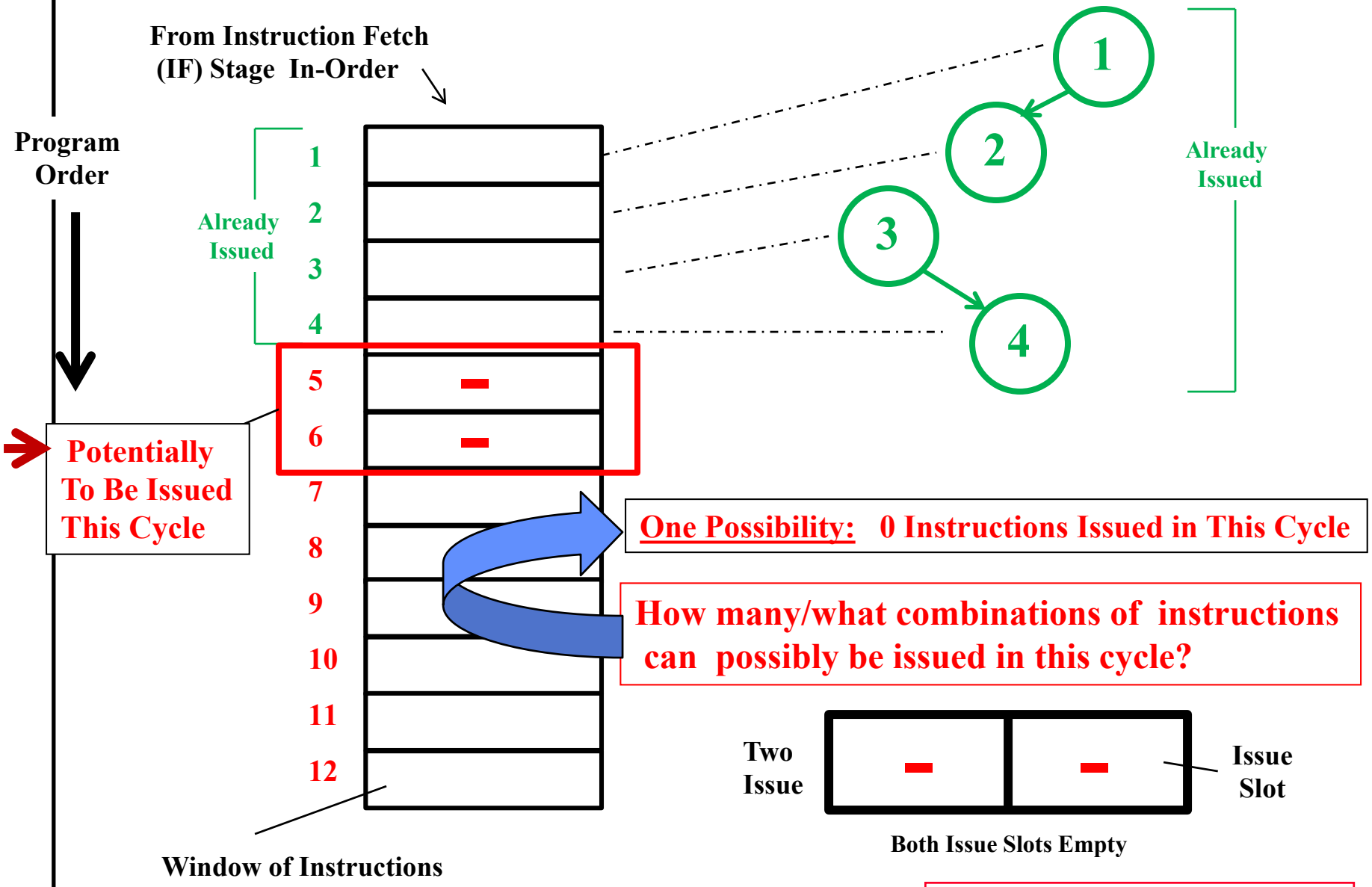


How many/what combinations of instructions can possibly be issued in this cycle?

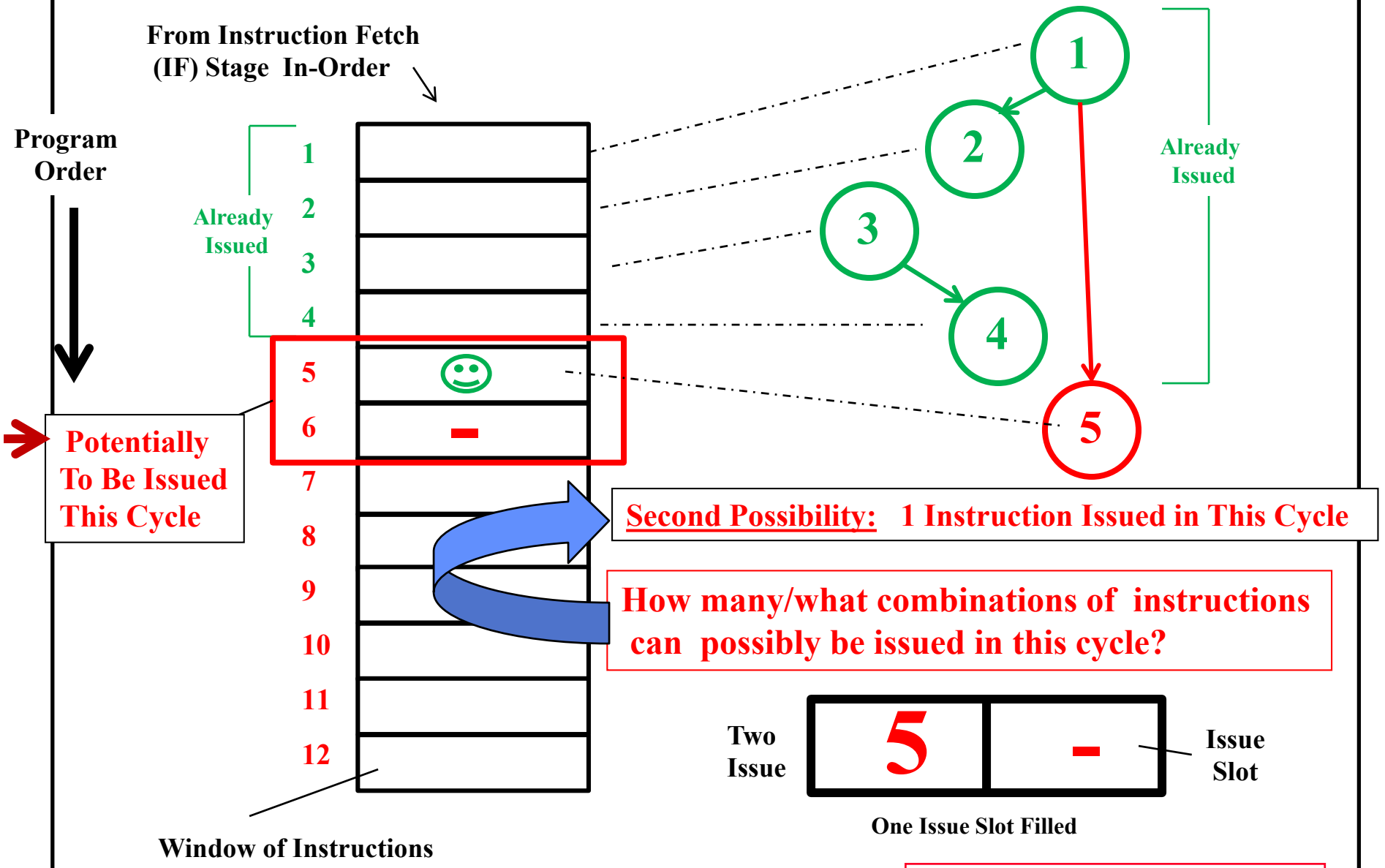
Multiple Instruction Issue Per Cycle:

The result of issuing multiple instructions in one cycle should be the same as if they were single-issued, one instruction per cycle.

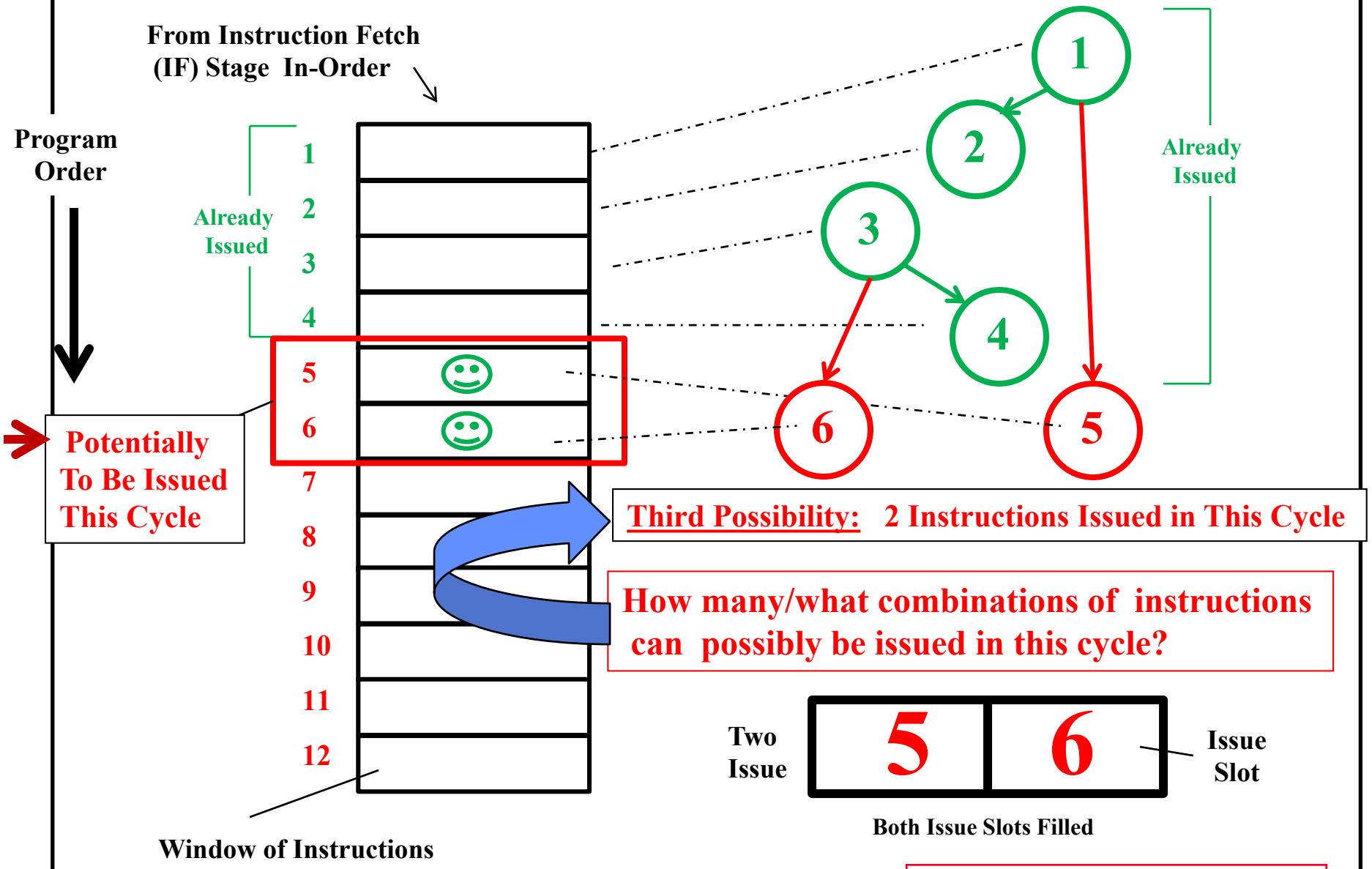
Two Instruction Issue Per Cycle (In-Order)



Two Instruction Issue Per Cycle (In-Order)



Two Instruction Issue Per Cycle (In-Order)



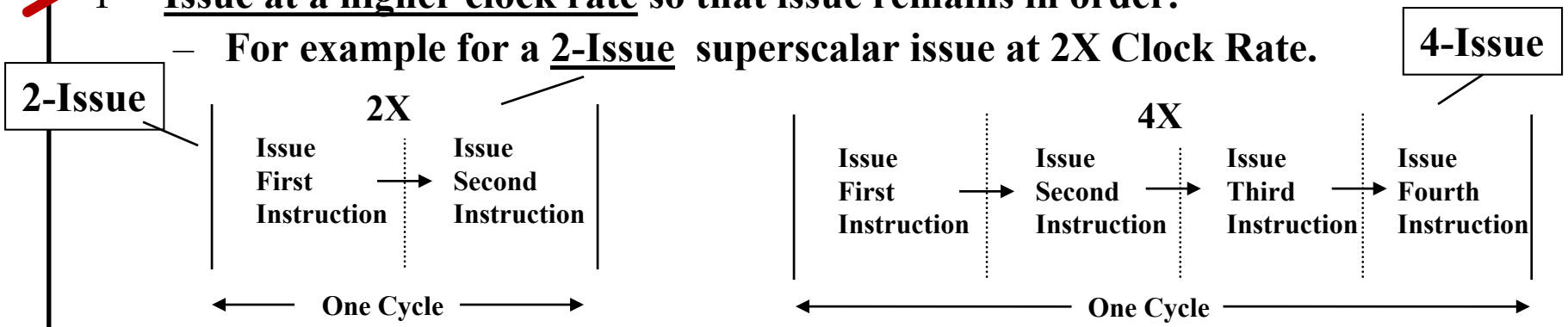
Superscalar Dynamic Scheduling

Three techniques can be used to support multiple instruction issue in Tomasulo

without putting restrictions on the type of instructions issued per cycle:

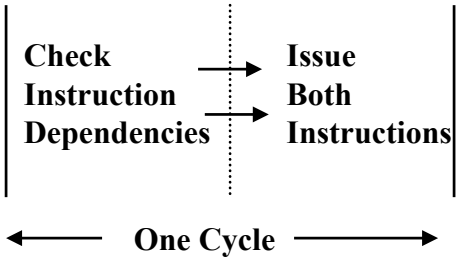
1 **Issue at a higher clock rate so that issue remains in order.**

- **For example for a 2-Issue superscalar issue at 2X Clock Rate.**



2 **Widen the issue logic to handle multiple instruction issue**

- **All possible dependencies between instructions to be issues are detected at once and the result of the multiple issue matches in-order issue**



2-Issue superscalar

0, 1 or 2 instructions issued per cycle for either method

Why?

For correct dynamic construction of dependency graph:
The result of issuing multiple instructions in one cycle should be the same as if they were single-issued, one instruction per cycle.

Superscalar Dynamic Scheduling

3 To avoid increasing the CPU clock cycle time in the last two approaches, multiple instruction issue can be spilt into two pipelined issue stages:

– Issue Stage One: Decide how many instructions can issue simultaneously checking dependencies only within the group of instructions to be issued + available RSs, ignoring instructions already issued.

– Issue Stage Two: Examine dependencies among the selected instructions from the group and the those already issued.

Add issued instructions to existing dependency graph

- This approach is usually used in dynamically-scheduled wide superscalars that can issue four or more instructions per cycle.
- Splitting the issue into two pipelined staged increases the CPU pipeline depth and increases branch penalties
  - This increases the importance of accurate dynamic branch prediction methods.
- Further pipelining of issue stages beyond two stages may be necessary as CPU clock rates are increased.
- The dynamic scheduling/issue control logic for superscalars is generally very complex *growing at least quadratically* with issue width.
  - e.g. 4 wide superscalar -> 4x4 = 16 times complexity of single issue CPU

Complexity

More control logic complexity/area/power

Multiple Instruction Issue with Dynamic Scheduling Example

- **The next example illustrates:**
  - – Multiple Issue Tomasulo.

+

  - – First method to handle/deal with a wrong dynamic branch prediction :

Delay start of execution of instructions following a branch until after the branch is resolved

Why? **In case of a branch misprediction (wrong prediction), such instructions with delayed start of execution did not update registers or memory and thus easy to cancel.**

Multiple Instruction Issue with Dynamic Scheduling Example

Integer	FP	← One Cycle

Assumptions:

1. **Restricted 2-way superscalar: 1 integer, 1 FP Issue Per Cycle**
2. **A sufficient number of reservation stations and FP units is available.**
3. **Total two integer units available:**
   One integer unit (for ALU, effective address)
   One integer unit for branch condition
4. **2 CDBs**
5. **Execution cycles:**
   Integer: 1 cycle
   Load: 2 cycles (1 ex + 1 mem)
   FP add: 3 cycles
6. **Any instruction following a branch cannot start execution until after branch condition is evaluated in EX (resolved)**

   Why?
7. **Branches are single issued,**
8. **no delayed branch,**
9. **perfect branch prediction**

Example Consider the execution of the following simple loop, which adds a scalar in F2 to each element of a vector in memory. Use a MIPS pipeline extended with Tomasulo's algorithm and with multiple issue:

```
Loop:   L.D      F0,0(R1)      ;F0=array element
        ADD.D    F4,F0,F2      ;add scalar in F2
        S.D      F4,0(R1)      ;store result
        DADDIU   R1,R1,#-8     ;decrement pointer
                               ;8 bytes (per DW)
        BNE      R1,R2,LOOP    ;branch R1!=R2
```

X(i) = X(i) + S

Assume that both a floating-point and an integer operation can be issued on every clock cycle, even if they are dependent. Assume one integer functional unit is used for both ALU operations and effective address calculations and a separate pipelined FP functional unit for each operation type. Assume that Issue and Write Results take one cycle each and that there is dynamic branch-prediction hardware and a separate functional unit to evaluate branch conditions. As in most dynamically scheduled processors, the presence of the Write Results stage means that the effective instruction latencies will be one cycle longer than in a simple in-order pipeline. Thus, the number of cycles of latency between a source instruction and an instruction consuming the result is one cycle for integer ALU operations, two cycles for loads, and three cycles for FP add. Create a table showing when each instruction issues, begins execution, and writes its result to the CDB for the first three iterations of the loop. Assume two CDBs and assume that branches single issue (no delayed branches) but that branch prediction is perfect. Also show the resource usage for the integer unit, the floating-point unit, the data cache, and the two CDBs.

3rd Edition:Example on page 221 (not in 4th Edition)

Three Loop Iterations on Restricted 2-way Superscalar Tomasulo

FP EX = 3 cycles

Iteration number	Instructions		In Order Issues at	(Start) Executes	Data Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for L.D
1	S.D	F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU	R1,R1,#-8	2	4		5	Wait for ALU
1	BNE	R1,R2,Loop	3	6			Wait for DADDIU
2	L.D	F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D	F4,F0,F2	4	10		13	Wait for L.D
2	S.D	F4,0(R1)	5	8	14		Wait for ADD.D
2	DADDIU	R1,R1,#-8	5	9		10	Wait for ALU
2	BNE	R1,R2,Loop	6	11			Wait for DADDIU
3	L.D	F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD.D	F4,F0,F2	7	15		18	Wait for L.D
3	S.D	F4,0(R1)	8	13	19		Wait for ADD.D
3	DAADIU	R1,R1,#-8	8	14		15	Wait for ALU
3	BNE	R1,R2,Loop	9	16			Wait for DADDIU

BNE Single Issue

BNE Single Issue

BNE Single Issue

Figure 3.25 The clock cycle of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline. The Write Result stage does not apply to either stores or branches, since they do not write any registers. We assume a result is written to the CDB at the end of the clock cycle it is available in. This figure also assumes a wider CDB. For L.D and S.D, the execution is effective address calculation. For branches, the execute cycle shows when the branch condition can be evaluated and the prediction checked; we assume that this can happen as early as the cycle after issue, if the operands are available. Any instructions following a branch cannot start execution until after the branch condition has been evaluated. We assume one memory unit, one integer pipeline, and one FP adder. If two instructions could use the same functional unit at the same point, priority is given to the "older" instruction. Note that the load of the next iteration performs its memory access before the store of the current iteration.

FP ADD has 3 execution cycles
Branches single issue

Only one CDB is actually needed in this case.

19 cycles to complete three iterations

For instructions after a branch: Execution starts after branch is resolved

Resource Usage Table for Example:

Clock number	Integer ALU	FP ALU	Data cache	CDB
2	1/ L.D			
3	1 / S.D		1/ L.D	
4	1 / DADDIU			1/ L.D
5		1 / ADD.D		1 / DADDIU
6				
7	2 / L.D			
8	2 / S.D		2 / L.D	1 / ADD.D
9	2 / DADDIU		1 / S.D	2 / L.D
10		2 / ADD.D		2 / DADDIU
11				
12	3 / L.D			
13	3 / S.D		3 / L.D	2 / ADD.D
14	3 / DADDIU		2 / S.D	3 / L.D
15		3 / ADD.D		3 / DADDIU
16				
17				
18				3 / ADD.D
19			3 / S.D	
20				

Figure 3.26 Resource usage table for the example shown in Figure 3.25. The entry in each box shows the opcode and iteration number of the instruction that uses the functional unit heading the column at the clock cycle corresponding to the row. Only a single CDB is actually required and that is what we show.

CDB # 2 ?

Not Needed

Only one CDB is actually needed/used in this case.

Multiple Instruction Issue with Dynamic Scheduling Example

3rd Edition: Example on page 223
(Not in 4th Edition)

Example Consider the execution of the same loop on a two-issue processor, but, in addition, assume that there are separate integer functional units for effective address calculation and for ALU operations. Create a table as in Figure 3.25 for the first three iterations of the same loop and another table to show the resource usage.

Answer Figure 3.27 shows the improvement in performance: The loop executes in 5 clock cycles less (11 versus 16 execution cycles). The cost of this improvement is both a separate address adder and the logic to issue to it; note that, in contrast to the earlier example, a second CDB is needed. As Figure 3.28 shows this example has a higher instruction execution rate but lower efficiency as measured by the utilization of the functional units.

Three factors limit the performance (as shown in Figure 3.27) of the two-issue dynamically scheduled pipeline:

1. There is an imbalance between the functional unit structure of the pipeline and the example loop. This imbalance means that it is impossible to fully use the FP units. To remedy this, we would need fewer dependent integer operations per loop. The next point is a different way of looking at this limitation.
2. The amount of overhead per loop iteration is very high: two out of five instructions (the `DADDIU` and the `BNE`) are overhead. In the next chapter we look at how this overhead can be reduced.
3. The control hazard, which prevents us from starting the next `L.D` before we know whether the branch was correctly predicted, causes a one-cycle penalty on every loop iteration. The next section introduces a technique that addresses this limitation.

Assumptions:

1. **The same loop in previous example**
2. **On restricted 2-way superscalar:**
   1 integer, 1 FP Issue Per Cycle
3. **A sufficient number of reservation stations and FP units is available.**
4. **Total three integer units** (One More)
   one for ALU, one for effective address
   One integer unit for branch condition
5. **2 CDBs**
6. **Execution cycles:**
   Integer: 1 cycle
   Load: 2 cycles (1 ex + 1 mem)
   FP add: 3 cycles
7. **Any instruction following a branch cannot start execution until after branch condition is evaluated**
8. **Branches are single issued,**
9. **no delayed branch,**
   perfect branch prediction

Previous example repeated with one more integer ALU (3 total)

Same three loop Iterations on Restricted 2-way Superscalar Tomasulo
but with Three integer units (one for ALU, one for effective address calculation, one for branch condition)

FP EX = 3 cycles

Iteration number	Instructions	In Order Issues at	(Start) Executes	Data Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	3	4	First issue
1	ADD.D F4,F0,F2	1	5		8	Wait for L.D
1	S.D F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1,R1,#-8	2	3		4	Executes earlier
1	BNE R1,R2,Loop	3	5			Wait for DADDIU
2	L.D F0,0(R1)	4	6	7	8	Wait for BNE complete
2	ADD.D F4,F0,F2	4	9		12	Wait for L.D
2	S.D F4,0(R1)	5	7	13		Wait for ADD.D
2	DADDIU R1,R1,#-8	5	6		7	Executes earlier
2	BNE R1,R2,Loop	6	8			Wait for DADDIU
3	L.D F0,0(R1)	7	9	10	11	Wait for BNE complete
3	ADD.D F4,F0,F2	7	12		15	Wait for L.D
3	S.D F4,0(R1)	8	10	16		Wait for ADD.D
3	DADDIU R1,R1,#-8	8	9		10	Executes earlier
3	BNE R1,R2,Loop	9	11			Wait for DADDIU

BNE Single Issue

BNE Single Issue

BNE Single Issue

Figure 3.27 The clock cycle of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline with separate functional units for integer ALU operations and effective address calculation, which also uses a wider CDB. The extra integer ALU allows the DADDIU to execute earlier, in turn allowing the BNE to execute earlier, and, thereby, starting the next iteration earlier.

3rd Edition:page 224
(not in 4th Edition

16 cycles here
vs. 19 cycles
(with two integer units)

Both CDBs are used here (in cycles 4, 8)

For instructions after a branch: Execution starts after branch is resolved

Resource Usage Table for Example:

Clock number	Integer ALU	Address adder	FP ALU	Data cache	CDB #1	CDB #2
2		1/ L.D				
3	1 / DADDIU	1 / S.D		1/ L.D		
4					1/L.D	1 / DADDIU
5			1 / ADD.D			
6	2 / DADDIU	2 / L.D				
7		2 / S.D		2 / L.D	2 / DADDIU	
8					1 / ADD.D	2 / L.D
9	3 / DADDIU	3 / L.D	2 / ADD.D	1 / S.D		
10		3 / S.D		3 / L.D	3 / DADDIU	
11					3 / L.D	
12			3 / ADD.D		2 / ADD.D	
13				2 / S.D		
14						
15						3 / ADD.D
16				3 / S.D		

Figure 3.28 Resource usage table for the example shown in Figure 3.27, using the same format as Figure 3.26.

Both CDBs are used here (in cycles 4, 8)

3rd Edition:page 225
(not in 4th Edition

What if dynamic branch prediction is wrong?

First method (discussed earlier):

→ **Delay start of execution of instructions following a branch until after the branch is resolved**

Why? In case of a branch misprediction (wrong prediction), such instructions with delayed start of execution did not update registers or memory and thus easy to cancel.

Second method (discussed next): →

Speculative Execution → Speculative Tomasulo Processor

→ **Can start execution of instructions following a branch before the branch is resolved** →

Further Reduction of Impact of Branches on Performance of Pipelined Processors:

What if dynamic branch prediction is wrong?

Speculation (Speculative Execution)

- Compiler ILP techniques (loop-unrolling, software Pipelining etc.) are not effective to uncover maximum ILP when branch behavior is not well known at compile time.
- Full exploitation of the benefits of dynamic branch prediction and further reduction of the impact of branches on performance can be achieved by using speculation:

Speculation: An instruction is executed before the processor knows that the instruction should execute to avoid control dependence stalls (i.e. branch not resolved yet):

- **Static Speculation** by the compiler with hardware support:

ISA/Compiler Support Needed

  - The compiler labels an instruction as speculative and the hardware helps by ignoring the outcome of incorrectly speculated instructions.
  - Conditional instructions provide limited speculation.

Done By CPU

- **Dynamic Hardware-based Speculation**:

4th Edition: Chapter 2.6, 2.8
(3rd Edition: Chapter 3.7)

No ISA or Compiler Support Needed

  - Uses dynamic branch-prediction to guide the speculation process.
  +
  - Dynamic scheduling and execution continued passed a conditional branch in the predicted branch direction.

e.g. dynamic speculative execution

Here we focus on hardware-based speculation using Tomasulo-based dynamic scheduling enhanced with speculation (Speculative Tomasulo Processor).

- The resulting processors are usually referred to as Speculative Processors.

Dynamic Hardware-Based Speculation

- **Combines:** (Speculative Execution Processors, Speculative Tomasulo)
  1. – Dynamic hardware-based branch prediction
  + 2. – Dynamic Scheduling: issue multiple instructions in order and execute out of order. (Tomasulo)
- **Continue to dynamically issue, and execute instructions passed a conditional branch in the dynamically predicted branch direction, before control dependencies are resolved.**

  i.e. before branch is resolved

  Why? – This overcomes the ILP limitations of the basic block size.
  - Creates dynamically speculated instructions at run-time with no ISA/compiler support at all. i.e. Dynamic speculative execution

  But – Branch mispredicted? If a branch turns out as mispredicted all such dynamically speculated instructions must be prevented from changing the state of the machine (registers, memory).

  i.e. speculated instructions must be cancelled

  ? How?
  - Addition of commit (retire, completion, or re-ordering) stage and forcing instructions to commit in their order in the code (i.e. to write results to registers or memory in program order).
  - **Precise exceptions** are possible since instructions *must commit in order.* *i.e instructions forced to complete (commit) in program order*

4th Edition: Chapter 2.6, 2.8 (3rd Edition: Chapter 3.7)

Four Steps of Speculative Tomasulo Algorithm

1. Issue — [1] (In-order) Get an instruction from Instruction Queue

   If [2] a reservation station and [3] a reorder buffer slot are free, issue instruction & send operands & reorder buffer number for destination (this stage is sometimes called "dispatch")

   → Stage 0 Instruction Fetch (IF): No changes, in-order

2. Execution — (out-of-order) Operate on operands (EX)

   Includes data MEM read (load)

   When both operands are ready then execute; if not ready, watch CDB for result; when both operands are in reservation station, execute; checks RAW (sometimes called "issue")

3. Write result — (out-of-order) Finish execution (WB)

   → No write to registers or data memory (store) in WB

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

   i.e Reservation Stations

   No WB for stores or branches

4. Commit — (In-order) Update registers, memory with reorder buffer result

   Data

   i.e. stores

   – When an instruction is at head of reorder buffer & the result is present, update register with result (or store to memory) and remove instruction from reorder buffer.

   → Successfully completed instructions write to registers and memory (stores) here

   Mispredicted Branch Handling

   – A mispredicted branch at the head of the reorder buffer flushes the reorder buffer (cancels all speculated instructions after the branch).

   ⇒ Instructions issue in order, execute (EX), write result (WB) out of order, but must commit in order.

Stages of Speculative Tomasulo Algorithm

Including Instruction Fetch (IF): No changes, still in-order

In program order

Can be reached out of program order
Including Start of Execution

In program order

In-Order Front End

Out-of-Order Core

In-Order Back End

IF → IS → EX → WB → Commit

Issue

Includes Data MEM Read (Loads)

- Decode instruction.
- Instruction issued in-order to a free reservation station(RS) and a reorder buffer slot or entry.
- No explicit "Read Operands" step:
  - Available instruction operands values supplied (from ISA registers) to assigned RS.
  - Operands not available yet are renamed to RSs that will produce the operand (register renaming).
  - (Dynamic construction of data dependency graph).

- Operate on operands.
- Also Includes data MEM read stage (loads).
- Stores do not write to data memory here.
- When all operands are ready then start executing on assigned FU.
- If all operands are not ready, watch Common Data Bus (CDB) for needed result (forwarding done via CDB). (i.e. wait on any remaining operands, no RAW)

- Write result on Common Data Bus (CDB) to all awaiting units (RSs) and reorder buffer entry.
- i.e. broadcast result on CDB (data forwarding).
- No WB for stores or branches.
- No write to registers or data memory in WB.

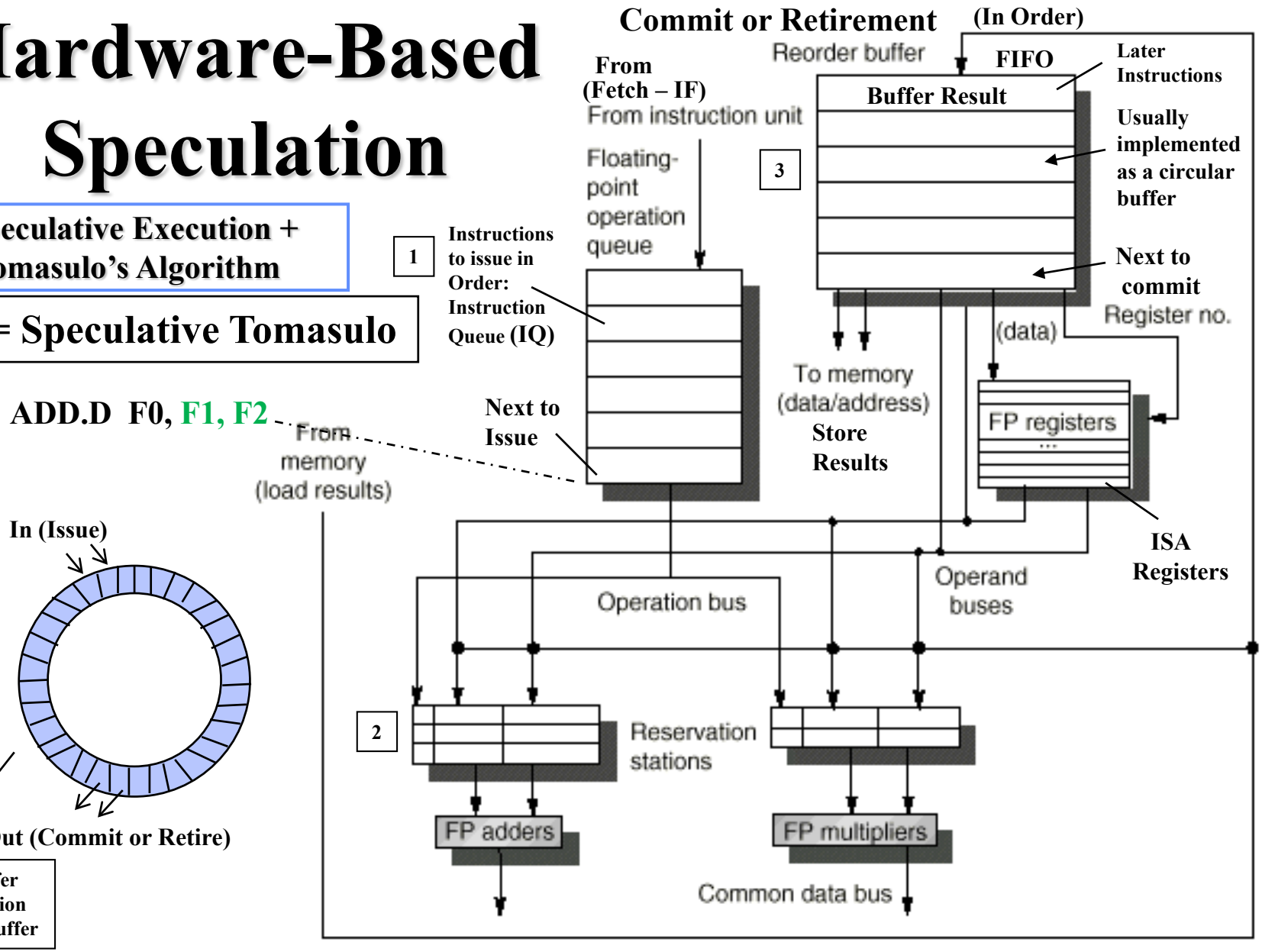
- When an instruction is at (reaches) head of reorder buffer & the result is present (buffered):
  - Update destination register, data memory (stores) with reorder buffer result.
- Successfully completed instructions write to registers and data memory (stores) here.
- Remove instruction from reorder buffer (instruction is completed).

Hardware-Based Speculation

Speculative Execution + Tomasulo's Algorithm

= Speculative Tomasulo

e.g. ADD.D F0, F1, F2



Circular Buffer Implementation of Reorder Buffer

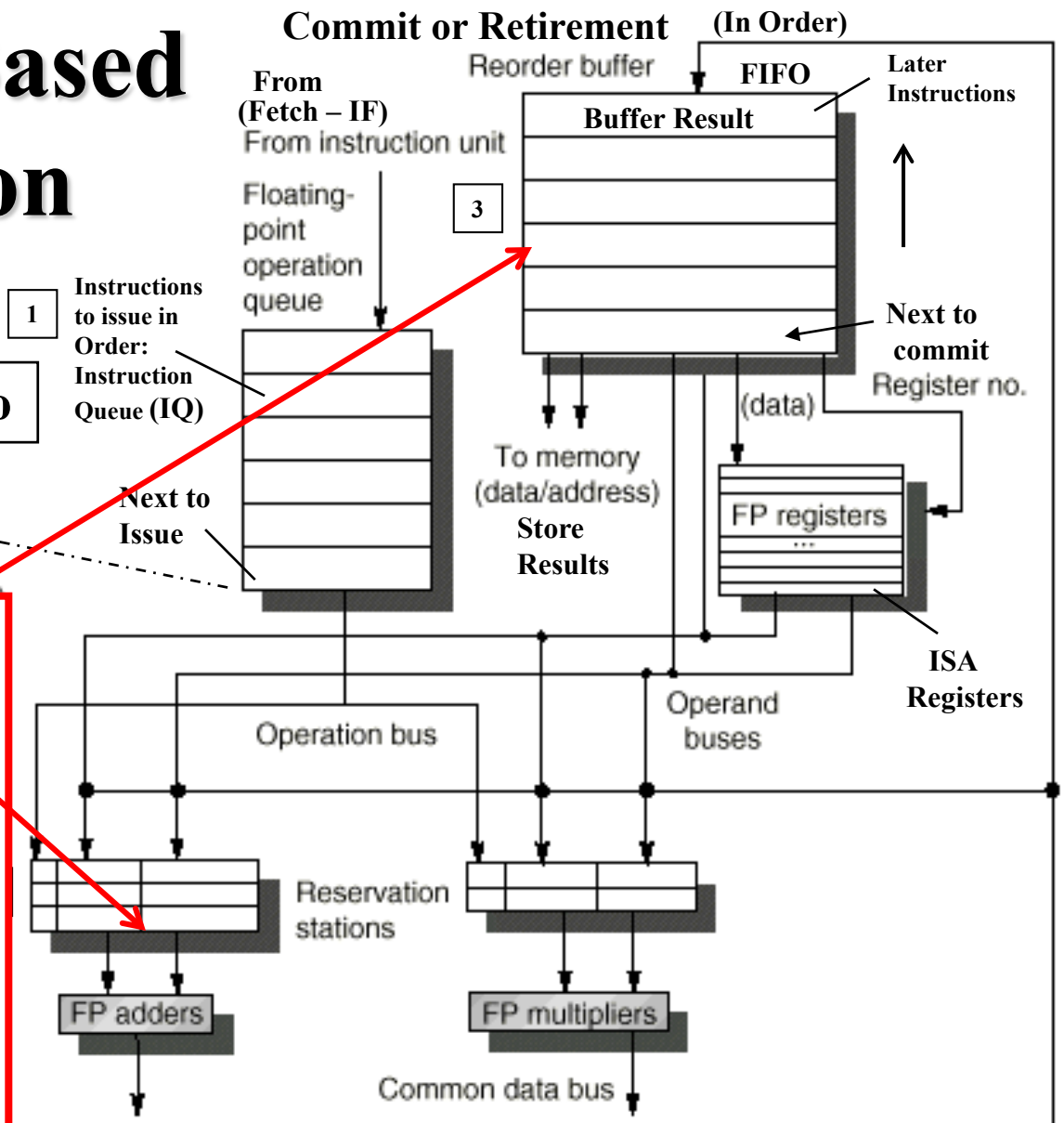
Speculative Tomasulo-based Processor

Hardware-Based Speculation

Speculative Execution + Tomasulo's Algorithm

= Speculative Tomasulo

e.g. ADD.D F0, F1, F2



Issue Instruction
To Reservation Station (RS)
ADD1
And allocate reorder buffer entry
The values of both
Operand registers F1, F2
Are read from register bank
Into V fields of RS: Vj, Vk

Since all operands are ready
Execution starts next cycle

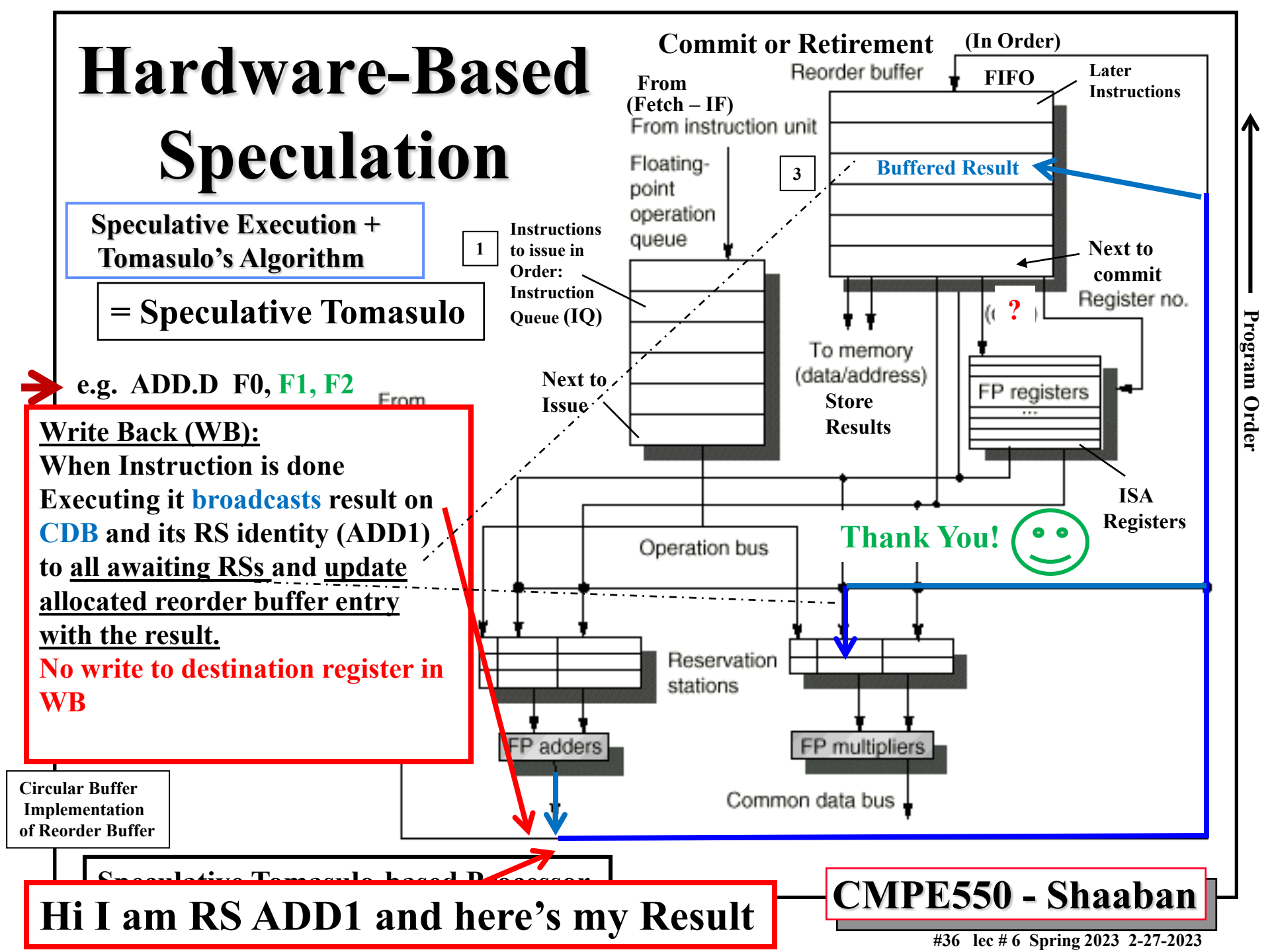
Speculative Tomasulo-based Processor

Hardware-Based Speculation

Speculative Execution + Tomasulo's Algorithm

= Speculative Tomasulo

e.g. ADD.D F0, F1, F2



Write Back (WB):
When Instruction is done Executing it broadcasts result on CDB and its RS identity (ADD1) to all awaiting RSs and update allocated reorder buffer entry with the result.
No write to destination register in WB

Circular Buffer Implementation of Reorder Buffer

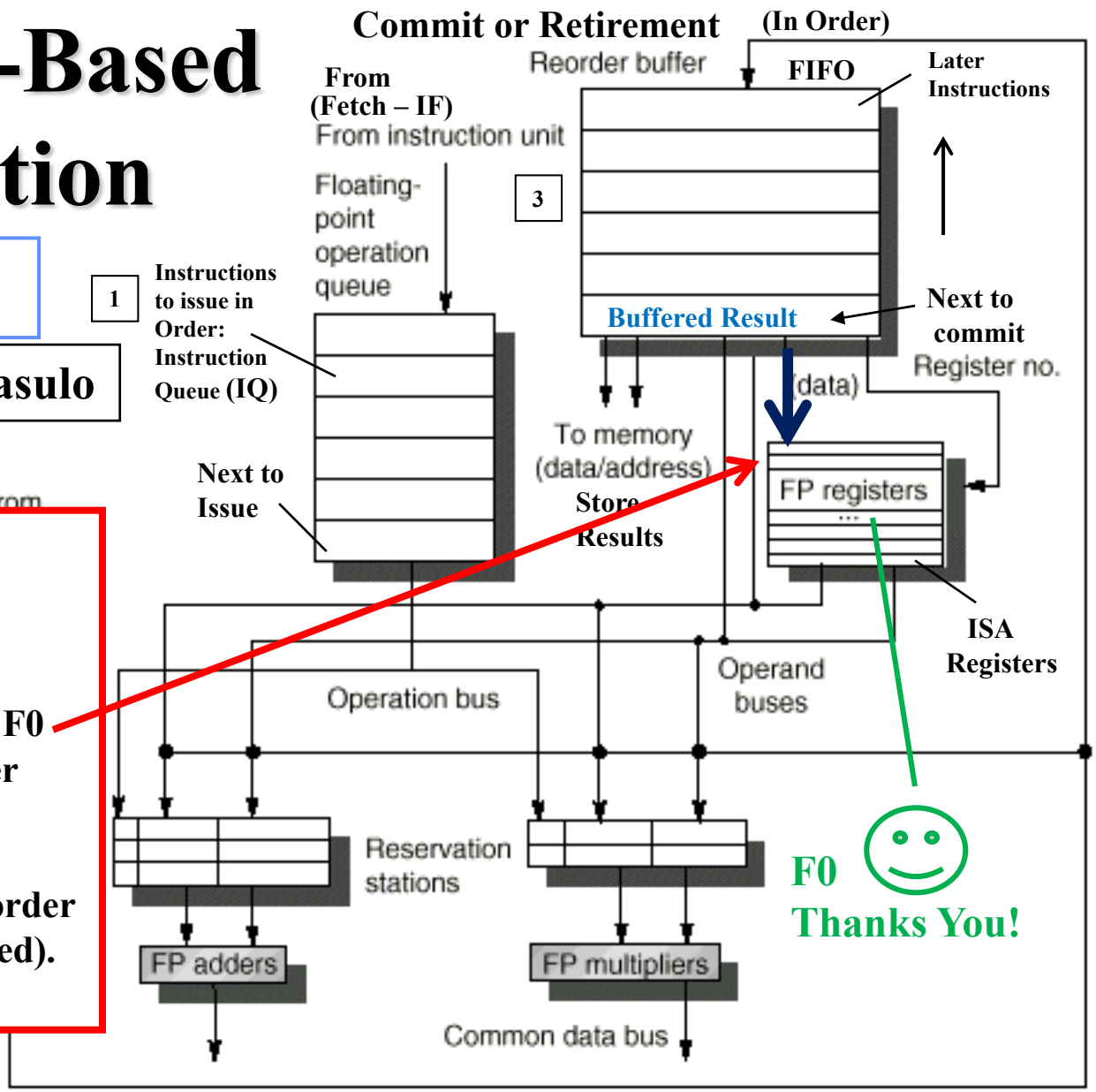
Hi I am RS ADD1 and here's my Result

Hardware-Based Speculation

Speculative Execution + Tomasulo's Algorithm

= Speculative Tomasulo

e.g. ADD.D F0, F1, F2



Commit:

- **When an instruction reaches head of reorder buffer:**
- **➢Update destination register F0 with result buffered in reorder buffer entry,**
- **Remove instruction from reorder buffer (instruction is completed).**

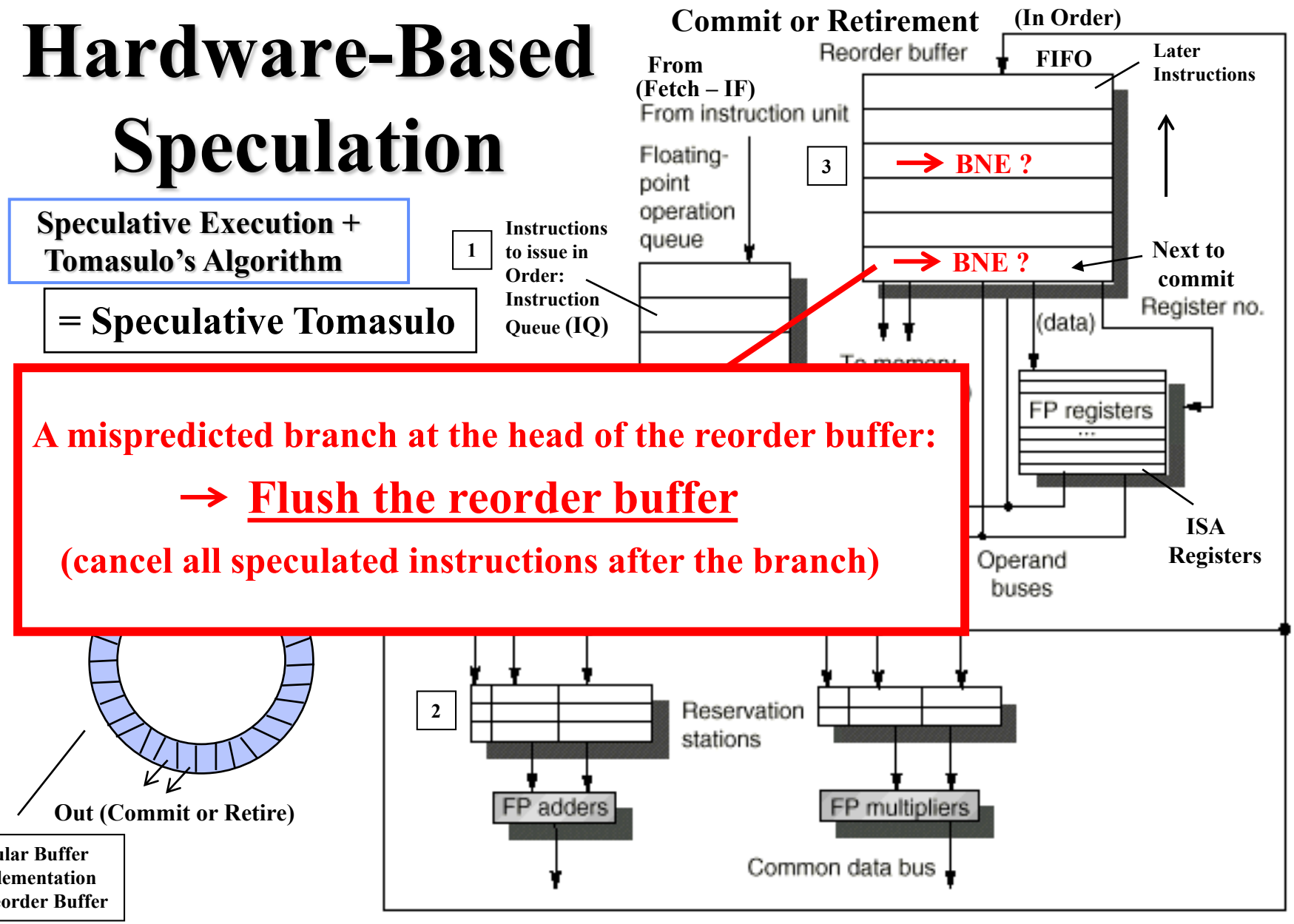
Implementation of Reorder Buffer

Speculative Tomasulo-based Processor

Hardware-Based Speculation

Speculative Execution + Tomasulo's Algorithm

= Speculative Tomasulo



Speculative Tomasulo-based Processor

Hardware-Based Speculation Example

Example Assume the same latencies for the floating-point functional units as in earlier examples: add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles. Using the code segment below, the same one we used to generate Figure 3.4, show what the status tables look like when the MUL.D is ready to go to commit.

Already Done

Program Order

Ready to Commit

```
L.D     F6,34(R2)
L.D     F2,45(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F6,F2
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

Show speculated single-issue Speculative Tomasulo status when MUL.D is ready to commit (commit done in program order)

Answer The result is shown in the three tables in Figure 3.30. Notice that although the SUB.D instruction has completed execution, it does not commit until the MUL.D commits. The reservation stations and register status field contain the same basic information that they did for Tomasulo's algorithm (see page 189 for a description of those fields). The differences are that reservation station numbers are replaced with ROB entry numbers in the Qj and Qk fields, as well as in the register status fields, and we have added the Dest field to the reservation stations. The Dest field designates the ROB number that is the destination for the result produced by this reservation station entry.

Single-issue speculative Tomasulo Example

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL.D	Mem[45 + Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV.D		Mem[34 + Regs[R2]]	#3		#5	

Reorder buffer (AKA *Commit* or *Retirement* buffer)

Entry	Busy	Instruction		State	Destination	Value
1	no	L.D	F6,34(R2)	Commit	F6	Mem[34 + Regs[R2]]
2	no	L.D	F2,45(R3)	Commit	F2	Mem[45 + Regs[R3]]
3	yes	MUL.D	F0,F2,F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D	F8,F6,F2	Write result	F8	#1 – #2
5	yes	DIV.D	F10,F0,F6	Execute	F10	
6	yes	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	. . .	yes	yes

Figure 3.30 **At the time the MUL.D is ready to commit, only the two L.D instructions have committed, although several others have completed execution.** The MUL.D is at the head of the ROB, and the two L.D instructions are there only to ease understanding. The SUB.D and ADD.D instructions will not commit until the MUL.D instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The DIV.D is in execution, but has not completed solely due to its longer latency than MUL.D. The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed, but are shown for informational purposes. We do not show the entries for the load-store queue, but these entries are kept in order.

Reorder buffer entry # 3 for MUL.D

Reorder buffer entry # 5 for DIV.D

Already committed

Program Order

Next to commit

Value: Buffered Result produced by instruction

speculated Tomasulo status when MUL.D is ready to commit (next cycle)

No result value in reorder buffer For DIV.D (not done executing)

Single-issue speculative Tomasulo Example

Hardware-Based Speculation Example

Example Consider the code example used earlier for Tomasulo's algorithm and shown in Figure 3.6 in execution:

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,#-8
        BNE      R1,R2,Loop    ;branches if R1≠R2
```

First iteration instructions committed

Next instruction to commit →

Assume that we have issued all the instructions in the loop twice. Let's also assume that the L.D and MUL.D from the first iteration have committed and all other instructions have completed execution. Normally, the store would wait in the ROB for both the effective address operand (R1 in this example) and the value (F4 in this example). Since we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

Answer The result is shown in the two tables in Figure 3.31.

Because neither the register values nor any memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted. Suppose that in the previous example (see Figure 3.31), the branch BNE is not taken the first time. The instructions prior to the branch will simply commit when each reaches the head of the ROB; when the branch reaches the head of that buffer, the buffer is simply cleared and the processor begins fetching instructions from the other path.

Single-issue speculative Tomasulo

4th Edition: pages 109-111 (3rd Edition page 231-232)

L.D. and MUL.D of first iteration have committed, other instructions completed execution

Already committed

Next to commit

Program Order

i.e. Instruction Result

Program Order

Reorder buffer

Entry	Busy	Instruction	State	Destination	Value
1	no	L.D F0,0(R1)	Commit	F0	Mem[0 + Regs[R1]]
2	no	MUL.D F4,F0,F2	Commit	F4	#1 × Regs[F2]
3	yes	S.D F4,0(R1)	Write result	0 + Regs[R1]	#2
4	yes	DADDIU R1,R1,#-8	Write result	R1	Regs[R1] – 8
5	yes	BNE R1,R2,Loop	Write result		
6	yes	L.D F0,0(R1)	Write result	F0	Mem[#4]
7	yes	MUL.D F4,F0,F2	Write result	F4	#6 × Regs[F2]
8	yes	S.D F4,0(R1)	Write result	0 + #4	#7
9	yes	DADDIU R1,R1,#-8	Write result	R1	#4 – 8
10	yes	BNE R1,R2,Loop	Write result		

What if branch was mispredicted ?

Speculative Instructions

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder #	6				7				
Busy	yes	no	no	no	yes	no	no	. . .	no

Figure 3.31 Only the L.D and MUL.D instructions have committed, although all the others have completed execution. Hence, no reservation stations are busy and none are shown. The remaining instructions will be committed as fast as possible. The first two reorder buffers are empty, but are shown for completeness.

Single-issue speculative Tomasulo

Multiple Issue (Superscalar) Tomasulo Example

- The next example → contrasts the operation of 2-issue Tomasulo **without** and **with** speculation:

→ – **Without Speculation:**

→ Delay start of execution of instructions following a branch until after the branch is resolved

→ – **With Speculation (2-Issue Speculative Tomasulo):**

→ Can start execution of instructions following a branch before the branch is resolved

+

→ Issue and Commit up to 2 instructions per cycle

→ **Branches still single issue (both)**

Multiple Issue with Speculation Example

(2-issue superscalar with no restriction on issue instruction type)

i.e issue up to 2 instructions and commit up to 2 instructions per cycle

Example Consider the execution of the following loop, which searches an array, on a two-issue processor, once without speculation and once with speculation:

```
Loop:   LD      R2,0(R1)        ;R2=array element
        DADDIU  R2,R2,#1        ;increment R2
        SD      R2,0(R1)        ;store result
        DADDIU  R1,R1,#4        ;increment pointer
        BNE     R2,R3,LOOP      ;branch if not last element
```

Integer code Ex = 1 cycle

2 CDBs

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table as in Figure 3.27 for the first three iterations of this loop for both machines. Assume that up to two instructions of any type can commit per clock.

Answer Figures 3.33 and 3.34 show the performance for a two-issue dynamically scheduled processor, without and with speculation. In this case, where a branch is a key potential performance limitation, speculation helps significantly. The third branch in the speculative processor executes in clock cycle 13, while it executes in clock cycle 19 on the nonspeculative pipeline. Because the completion rate on the nonspeculative pipeline is falling behind the issue rate rapidly, the nonspeculative pipeline will stall when a few more iterations are issued. The performance of the nonspeculative processor could be improved by allowing load instructions to

Other Assumptions:

- **A sufficient number of reservation stations and reorder (commit) buffer entries are available.**
- **Branches still single issue**

+ A sufficient number of Functional Units (FUs)/ALUs

Answer: Without Speculation

No Speculation: Delay execution of instructions following a branch until after the branch is resolved

Program Order

Iteration number	Instructions		In Order: Issues at clock cycle number	Executes at clock cycle number	Data Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD	R2,0(R1)	1	2	3	4	First issue
1	DADDIU	R2,R2,#1	1	5		6	Wait for LW
1	SD	R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU	R1,R1,#4	2	3		4	Execute directly
1	BNE	R2,R3,LOOP	3	7			Wait for DADDIU
2	LD	R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU	R2,R2,#1	4	11		12	Wait for LW
2	SD	R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU	R1,R1,#4	5	8		9	
2	BNE	R2,R3,LOOP	6	13			Wait for DADDIU
3	LD	R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU	R2,R2,#1	7	17		18	Wait for LW
3	SD	R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU	R1,R1,#4	8	14		15	Wait for BNE
3	BNZ	R2,R3,LOOP	9	19			Wait for DADDIU

BNE Single Issue

BNE Single Issue

BNE Single Issue

Figure 3.33 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the L.D following the BNE cannot start execution earlier, because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch condition evaluation allow multiple instructions to execute in the same cycle.

19 cycles to complete three iterations

Branches Still Single Issue

For instructions after a branch: Execution starts after branch is resolved

(No speculation)

Answer: 2-way Superscalar Tomasulo With Speculation

With Speculation:
Start execution of instructions following a branch before the branch is resolved

2-way Speculative Superscalar Processor: Issue and commit up to 2 instructions per cycle

2 CDBs

Program Order ↓

Iteration number	Instructions		Issues at clock number (In Order)	Executes at clock number	Loads — Data Memory Read access at clock number	Write CDB at clock number	Commits at clock number (In Order)	Comment
1	LD	R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU	R2,R2,#1	1	5		6	7	Wait for LW
1	SD	R2,0(R1)	2	3	-		7	Wait for DADDIU
1	DADDIU	R1,R1,#4	2	3		4	8	Commit in order
1	BNE	R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD	R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU	R2,R2,#1	4	8		9	10	Wait for LW
2	SD	R2,0(R1)	5	6	-		10	Wait for DADDIU
2	DADDIU	R1,R1,#4	5	6		7	11	Commit in order
2	BNE	R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD	R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU	R2,R2,#1	7	11		12	13	Wait for LW
3	SD	R2,0(R1)	8	9	-		13	Wait for DADDIU
3	DADDIU	R1,R1,#4	8	9		10	14	Executes earlier
3	BNE	R2,R3,LOOP	9	13			14	Wait for DADDIU

BNE Single Issue

BNE Single Issue

BNE Single Issue

Figure 3.34 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation. Note that the L.D following the BNE can start execution early because it is speculative.

Branches Still Single Issue

14 cycles here (with speculation) vs. 19 without speculation

Also see sample quiz 5 in myCourses

Arrows show data dependencies

Quiz # 5

On Lecture Notes Set # 6

- **Monday, March 22**

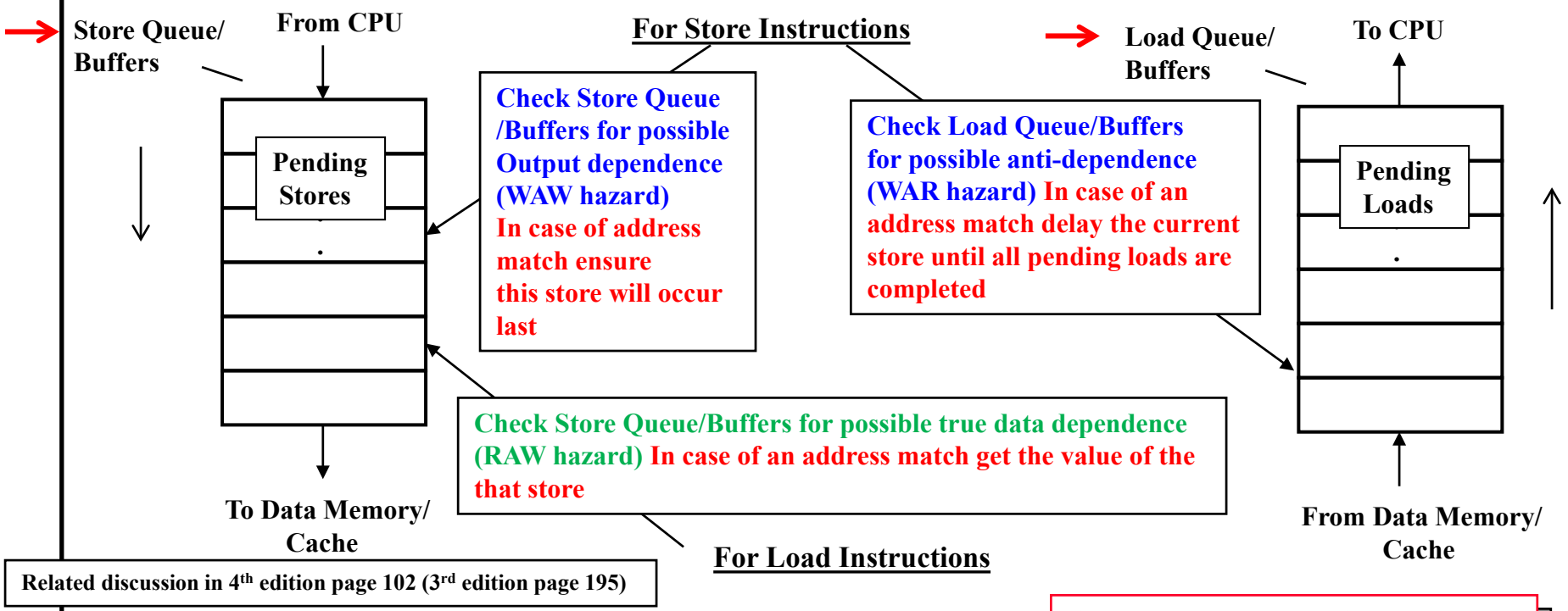
- **Quiz On Speculative Multiple-Issue Tomasulo (Including sample Quiz 5 covered in class)**

- **You may consult sample Quiz 5 while taking the quiz**

Data Memory Access Dependency Checking/Handling In Dynamically Scheduled Processors

i.e data + name

- Renaming in Tomasolu-based dynamically scheduled processors eliminates name dependence for register access but not for data memory access (loads and stores).
- Thus both true data dependencies and name dependencies must be detected to ensure correct ordering of data memory accesses for correct execution.
- One possible solution:
- For Loads: Check store queue/buffers to ensure no data dependence violation (RAW hazard)
- For Stores:
  - Check store queue/buffers to ensure no output name dependence violation (WAW hazard)
  - Check load queue/buffers to ensure no anti-dependence violation (WAR hazard)



Related discussion in 4th edition page 102 (3rd edition page 195)

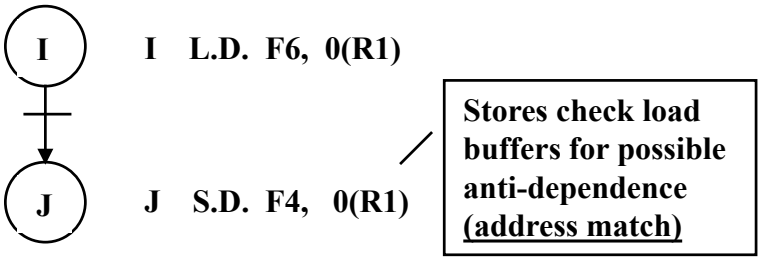
Note: Since instructions issue in program order, all pending load/store instructions in load/store queues are before the current load/store instruction in program order.

Data Memory Access Dependency Checking/Handling In Dynamically Scheduled Processors: Examples

Data Access Name Dependency Examples:

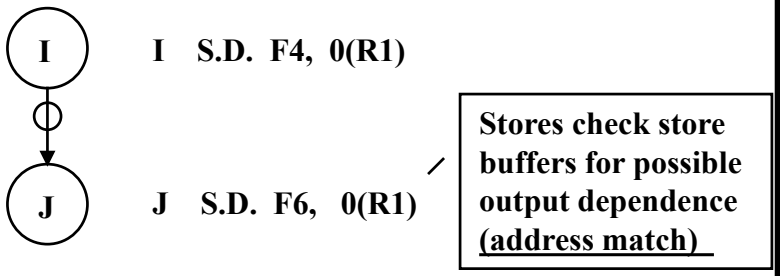
Assume value of R1 is not changed

Anti-dependence Example:



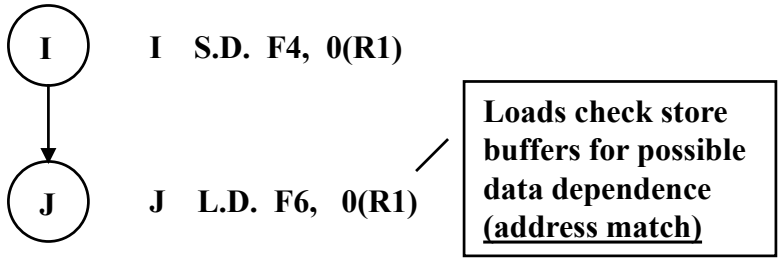
We have an address match here:
Instruction J (S.D.) must occur after I (i.e. write to memory by J must occur after read from memory by I) to prevent anti-dependence violation (WAR hazard).

Output-dependence Example:

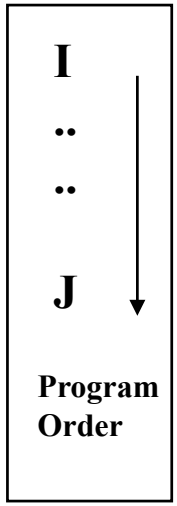


We have an address match here:
Instruction J (Second S.D.) must occur last (i.e. write to memory last) to prevent output-dependence violation (WAW hazard).

True Data Dependence Example:



We have an address match here:
Instruction J (L.D.) gets the value of I (S.D) from store buffer to prevent data dependence violation (RAW hazard).



What about memory access dependency checking in speculative Tomasulo?

Limits to Multiple Instruction Issue Processors

- **Inherent limitations of ILP:**
  - If 1 branch exist for every 5 instruction : How to keep a 5-way superscalar/VLIW processor busy?
  - Latencies of unit adds complexity to the many operations that must be scheduled every cycle.
  - For maximum performance multiple instruction issue requires about:

    Pipeline Depth x No. Functional Units

    active instructions at any given cycle.

    i.e to achieve 100% Functional unit utilization

- **Hardware implementation complexities:**
  - Duplicate FUs for parallel execution are needed, more CDBs.
  - More instruction bandwidth is essential.
  - Increased number of ports to Register File (datapath bandwidth):
    - VLIW example needs 7 read and 3 write for Int. Reg. & 5 read and 3 write for FP reg
  - Increased ports to memory (to improve memory bandwidth).
  - Superscalar issue/decoding complexity may impact pipeline clock rate, depth.

Superscalar Architecture Limitations:

Slide 13 Repeated Here

i.e. Multiple-issue processors

Issue Slot Waste Classification

- Empty or wasted issue slots can be defined as either vertical waste or horizontal waste:
  - – Vertical waste is introduced when the processor issues no instructions in a cycle. Full Stall?
  - – Horizontal waste occurs when not all issue slots can be filled in a cycle. Partial Stall?

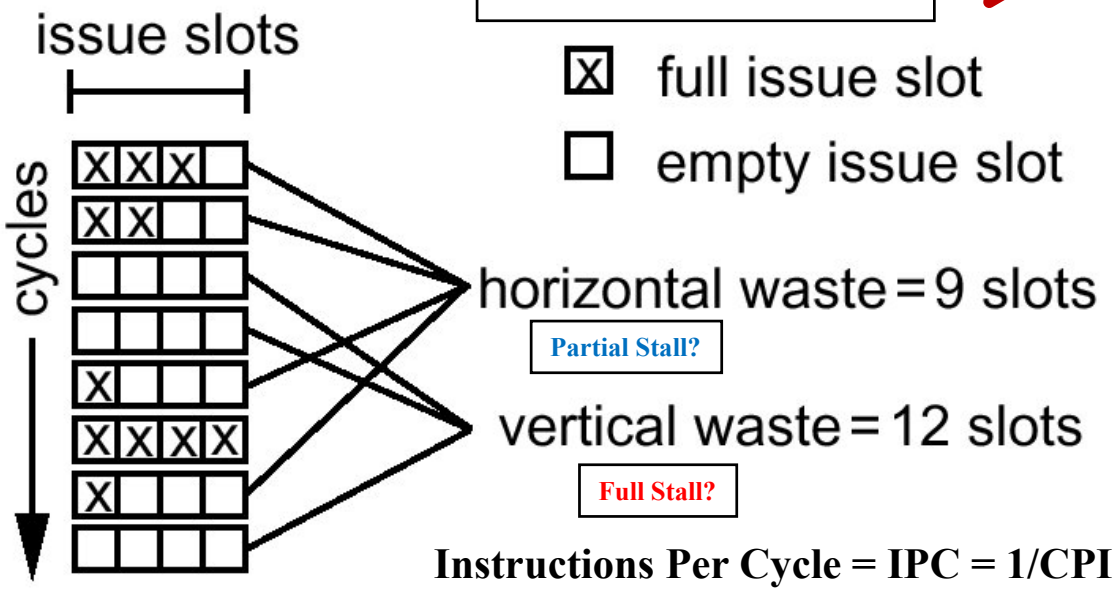
How About 8-Issue?

Example:

4-Issue Superscalar (Or VLIW)

Ideal IPC =4
Ideal CPI = .25

Time



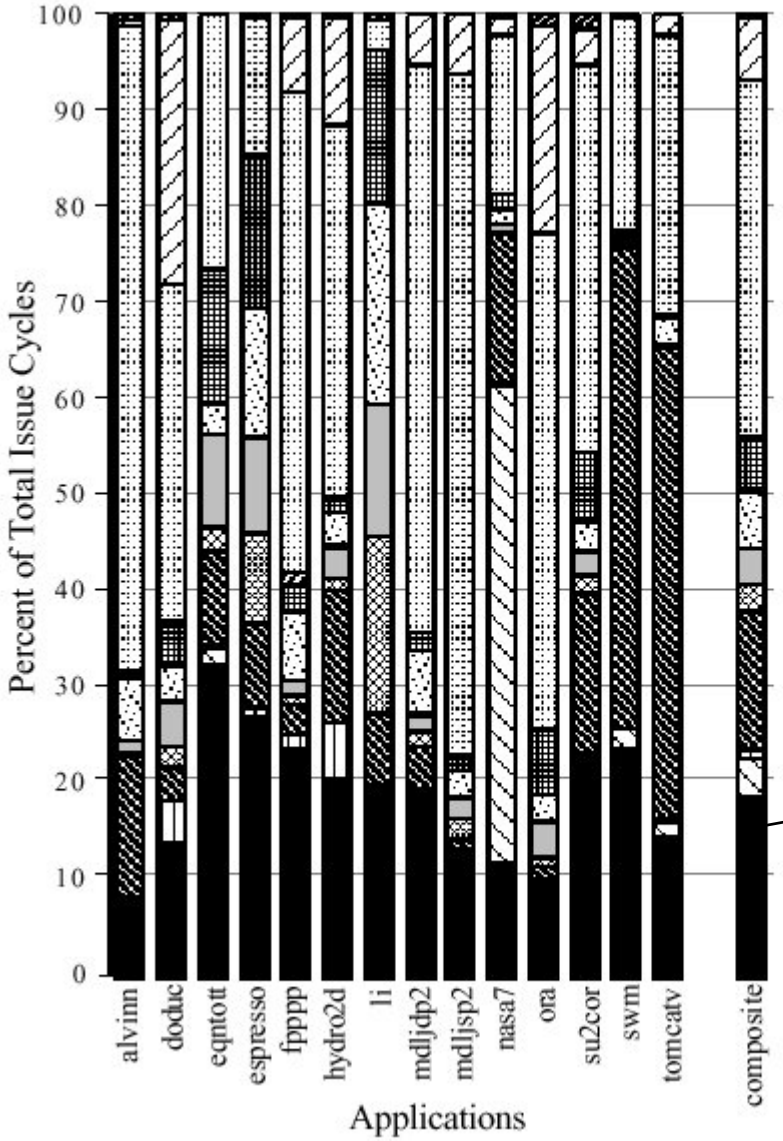
Also applies to VLIW

Instructions Per Cycle = IPC = 1/CPI

Result of issue slot waste: Actual Performance << Peak Performance

Sources of Unused Issue Cycles in an 8-issue Superscalar Processor

(wasted or empty)



Ideal Instructions Per Cycle, IPC = 8 (CPI = 1/8= 0.125)
Here real IPC about 1.5

(18.75 % of ideal IPC)

Real IPC << Ideal IPC

1.5 << 8

~ 81% of issue slots wasted (empty)

~ 19% of issue slots filled (Busy

***Processor busy* represents the utilized issue slots; all others represent wasted issue slots.**

61% of the wasted cycles are vertical waste, the remainder are horizontal waste.

Workload: SPEC92 benchmark suite.

Source: Simultaneous Multithreading: Maximizing On-Chip Parallelism Dean Tullsen et al., Proceedings of the 22rd Annual International Symposium on Computer Architecture, June 1995, pages 392-403.

Superscalar Architecture Limitations :

All possible causes of wasted issue slots, and latency-hiding or latency reducing techniques that can reduce the number of cycles wasted by each cause.

Source of Wasted Issue Slots	Possible Latency-Hiding or Latency-Reducing Technique
instruction tlb miss, data tlb miss	decrease the TLB miss rates (e.g., increase the TLB sizes); hardware instruction prefetching; hardware or software data prefetching; faster servicing of TLB misses
I cache miss	larger, more associative, or faster instruction cache hierarchy; hardware instruction prefetching
D cache miss	larger, more associative, or faster data cache hierarchy; hardware or software prefetching; improved instruction scheduling; more sophisticated dynamic execution
branch misprediction	improved branch prediction scheme; lower branch misprediction penalty
control hazard	speculative execution; more aggressive if-conversion
load delays (first-level cache hits)	shorter load latency; improved instruction scheduling; dynamic scheduling
short integer delay	improved instruction scheduling
long integer, short fp, long fp delays	(multiply is the only long integer operation, divide is the only long floating point operation) shorter latencies; improved instruction scheduling
memory conflict	(accesses to the same memory location in a single cycle) improved instruction scheduling

Main Issue: One Thread leads to limited ILP (cannot fill issue slots)

Solution: Exploit Thread Level Parallelism (TLP) within a single microprocessor chip:

How?

Simultaneous Multithreaded (SMT) Processor:

-The processor issues and executes instructions from a number of threads creating a number of logical processors within a single physical processor
e.g. Intel's HyperThreading (HT), each physical processor executes instructions from two threads

SMT +

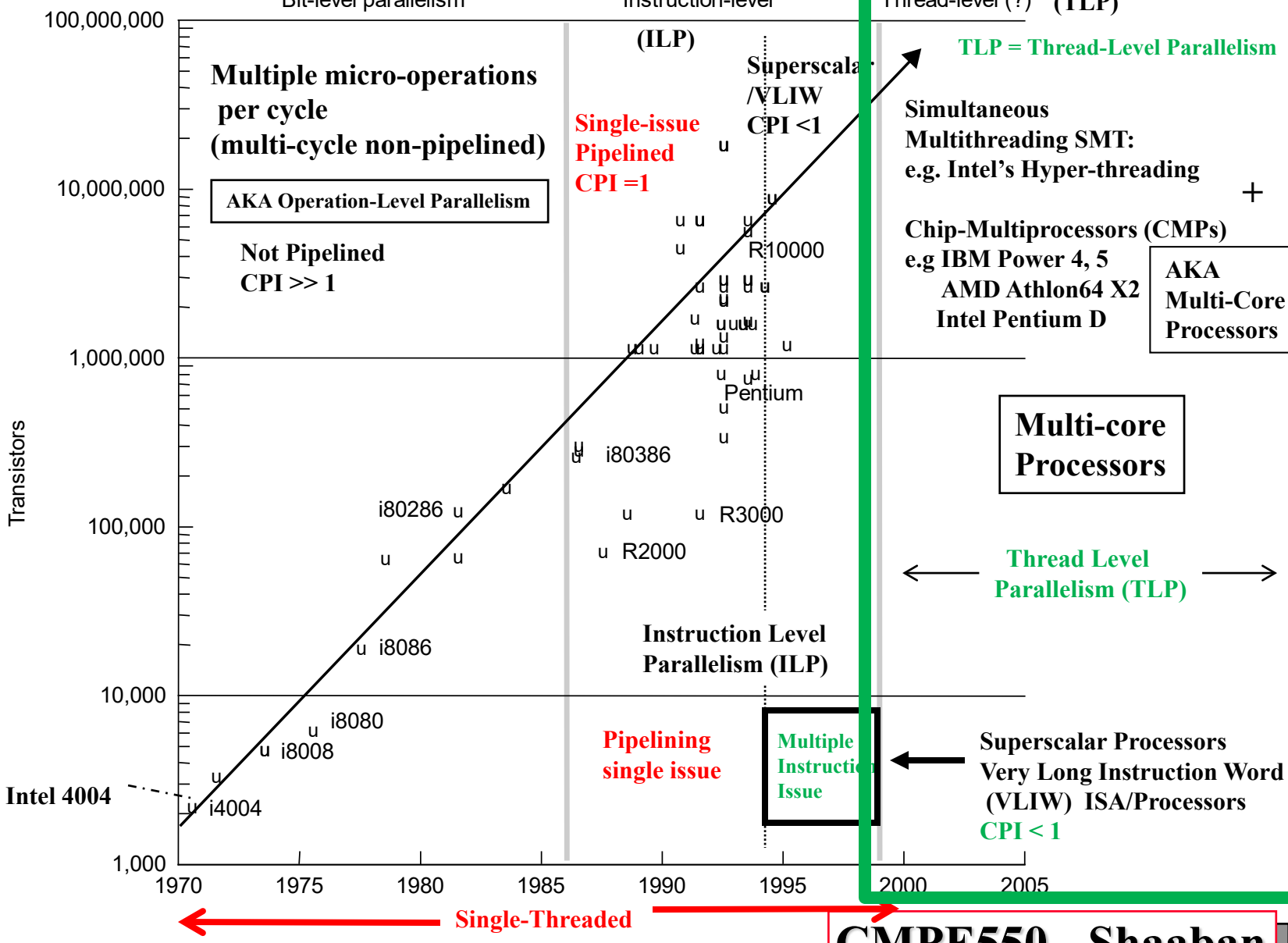
AND/OR

Chip-Multiprocessors (CMPs): **CMPs**

- Integrate two or more complete processor cores on the same chip (die)
- Each core runs a different thread (or program)
- Limited ILP is still a problem in each core
(Solution: combine this approach with SMT)

Source: Simultaneous Multithreading: Maximizing On-Chip Parallelism Dean Tullsen et al., Proceedings of the 22rd Annual International Symposium on Computer Architecture, June 1995, pages 392-403.

Parallelism in Microprocessor VLSI Generations



Dual-Core Chip-Multiprocessor (CMP) Architectures

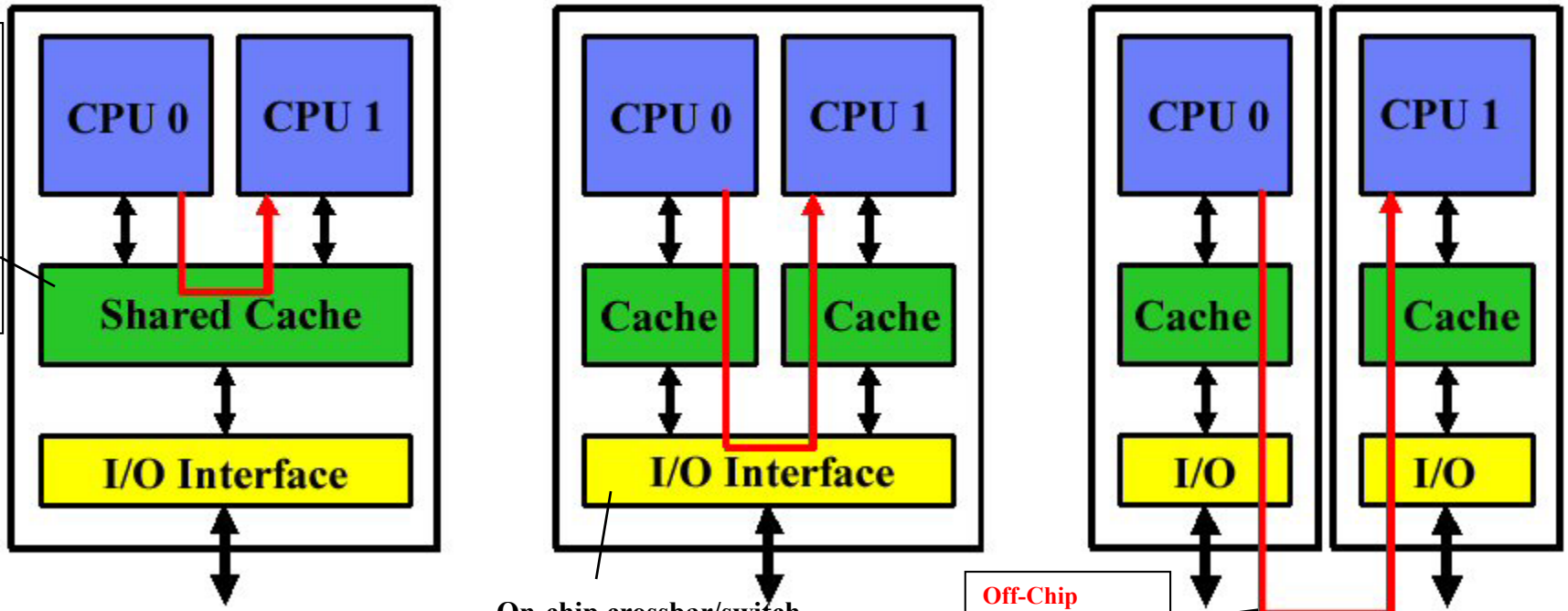
AKA "Multi-Core Processors

Single Die
Shared L2 or L3 Cache
(Native CMP Design?)

Single Die
Private Caches
Shared System Interface

Two Dies – Shared Package
Private Caches
Private System Interface

Level 2 (L2) or Level 3 (L3) Cache Shared Among Cores



Cores communicate using shared cache
(Lowest communication latency)

Examples:
IBM POWER4/5
Intel Pentium Core Duo (Yonah), Conroe (Core 2), Sun UltraSparc T1 (Niagara)
AMD Barcelona (quad-core) ……

On-chip crossbar/switch

Cores communicate using on-chip Interconnects (shared system interface)

Examples:
AMD Dual Core Opteron,
Athlon 64 X2
Intel Itanium2 (Montecito)
And Many-Core CMPs

Off-Chip Communication

FSB

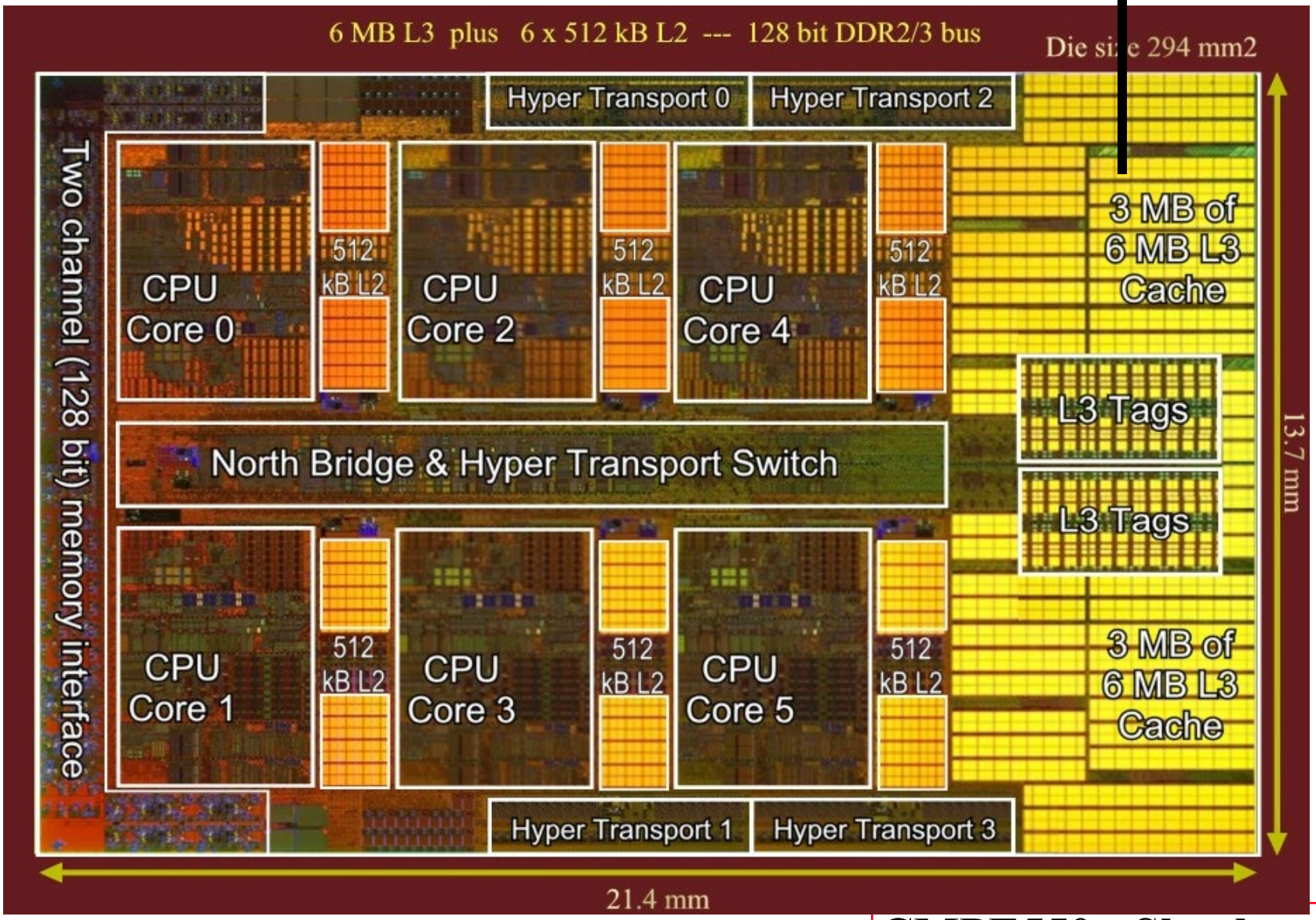
Cores communicate over external Front Side Bus (FSB)
(Highest communication latency)

Example:
Intel Pentium D, Core2 Quad cores

Source: Real World Technologies,
http://www.realworldtech.com/page.cfm?ArticleID=RWT101405234615

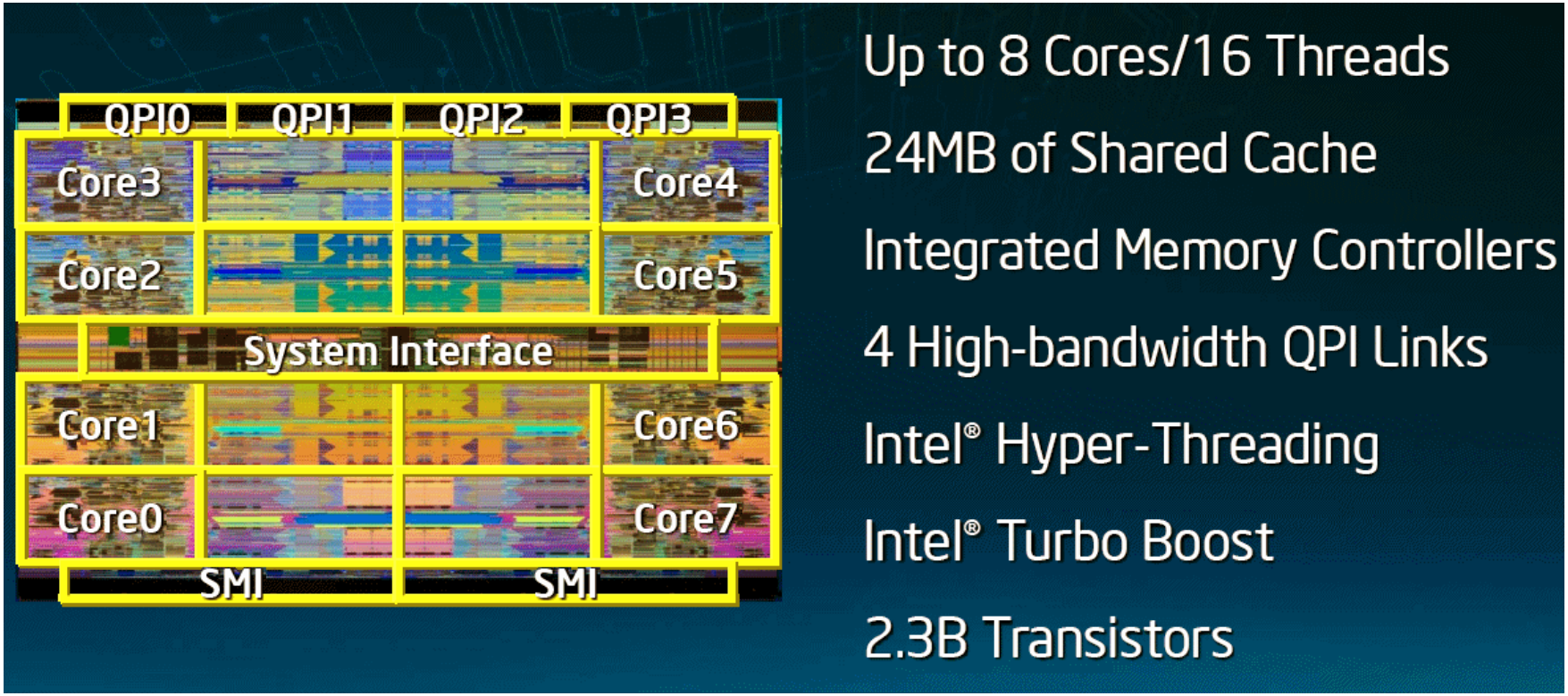
Example Six-Core Processor: AMD Phenom II X6

→ *Six processor cores sharing 6 MB of level 3 (L3) cache*



Example Eight-Core CMP: Intel Nehalem-EX

- *Eight processor cores sharing 24 MB of level 3 (L3) cache*
- *Each core is 2-way SMT (2 threads per core), for a total of 16 threads*

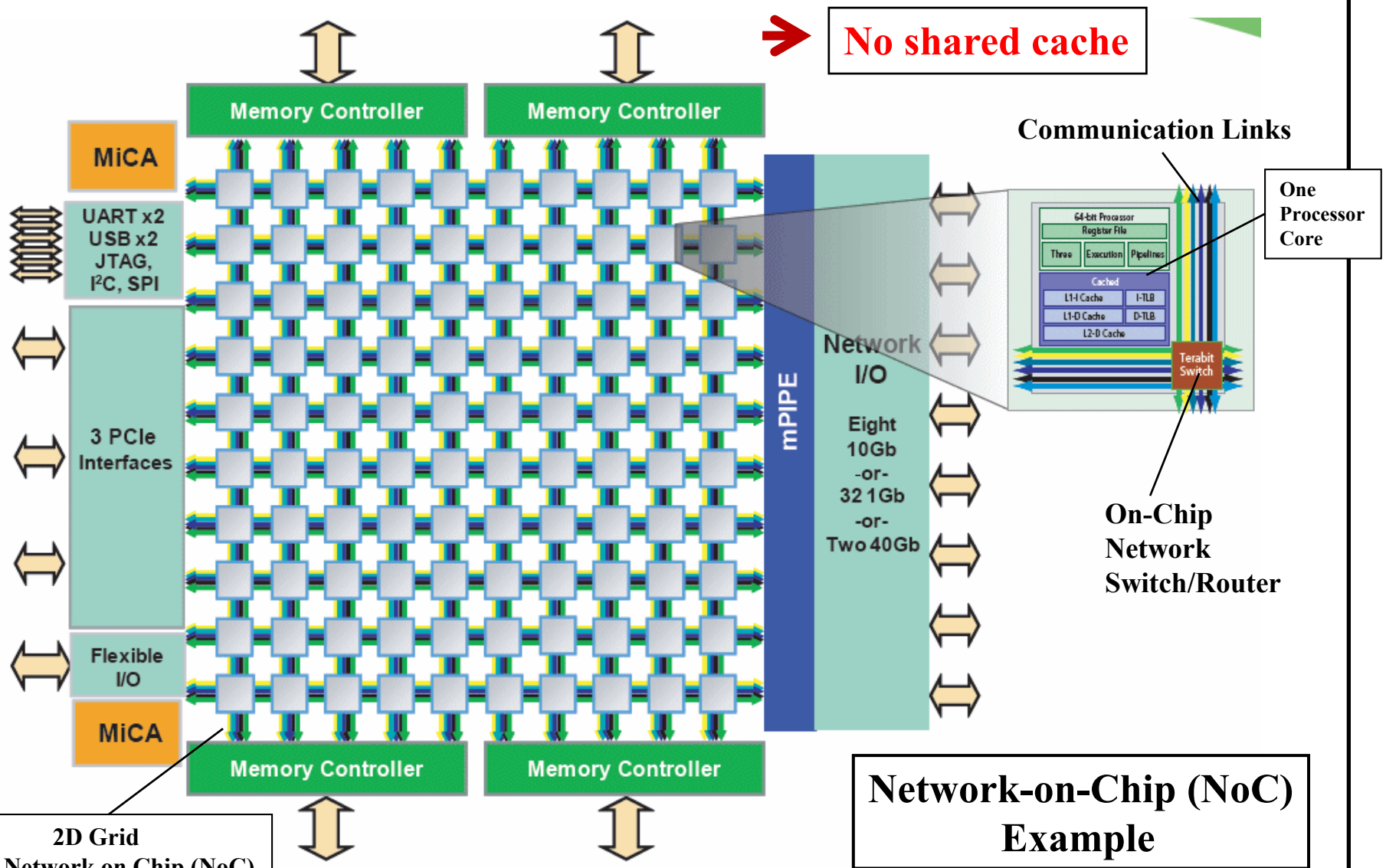


CMP = Chip-Multiprocessor
AKA Multi-Core Processor

Many-Core CMP?

Previously Tilera

Example 100-Core CMP: Mellanox TILE-Gx Processor



Network-on-Chip (NoC) Example

For more information see: http://www.tilera.com/

Quiz # 5

On Lecture Notes Set # 6

- **Monday, March 22**

- **Quiz On Speculative Multiple-Issue Tomasulo (Including sample Quiz 5 covered in class)**

- **You may consult sample Quiz 5 while taking the quiz**