

CS4223

Multi-Core Architectures

Trevor E. Carlson
National University of Singapore
tcarlson@comp.nus.edu.sg
(Slides from Tulika Mitra)

Feedback

- “According to NUSMods, there is 1hr tutorial after 2hr lecture. I believe we could better utilise the time allocated.
- How would the tutorials be like? Will it be similar to lecture? Please share how you would make full use of the time.”

Feedback

- “Dispite the number of rules and regulations, it is great that there is much engagement in class!”
- “Thanks for your hard work, even to those who cannot make it to class. Keep up the good job.”

Feedback

- “[...] it would be nice if you can put up some important questions or problem statements that are relevant to the content of the lectures and are still open in the research community.

You never know which problem catches the attention of which student and may be he/she has some interesting perspective about it, that might lead to a possible project in the future. ” – CS4223 Student

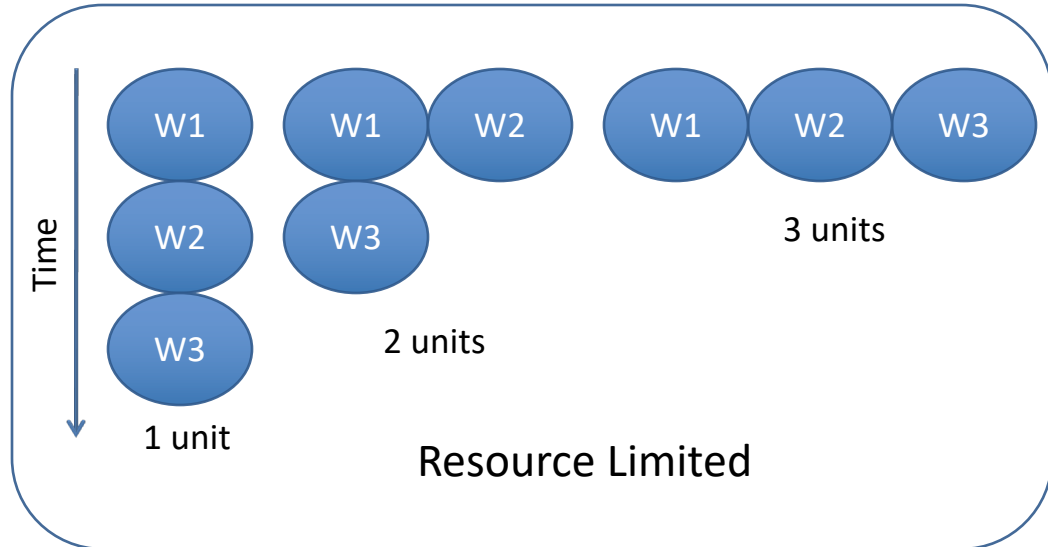
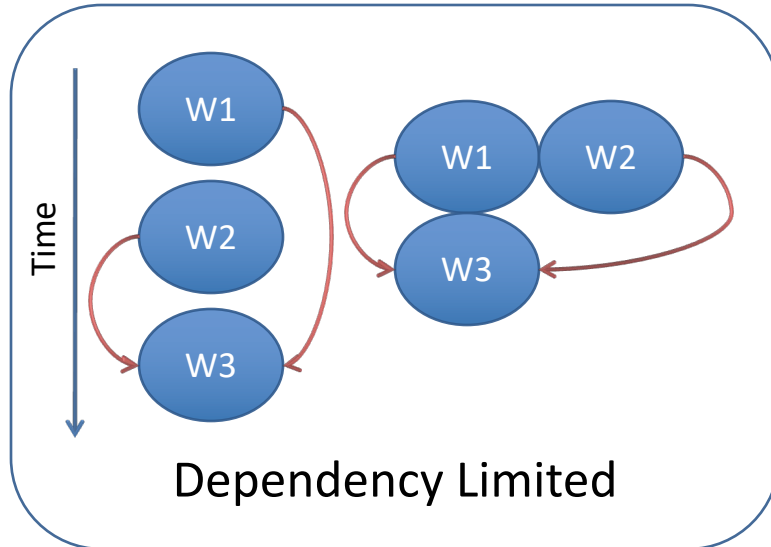
Limits of ILP

Table 2: Benchmark IPC for All Configurations

Benchmark	IPC No Renaming	IPC Register Renaming	IPC Memory Renaming	IPC r29 Removed	IPC 10K Window
compress95	3.12	26.25	73.88	226.33	18.89
cc1	3.61	39.79	41.63	239.96	86.45
go	2.50	49.15	53.77	141.46	70.71
ijpeg	2.41	55.47	93.60	94.11	52.94
li	3.56	19.60	19.61	81.45	27.70
m88ksim	2.76	19.93	62.06	363.26	20.50
perl	3.47	82.01	127.57	153.05	128.84
vortex	4.57	26.26	26.27	271.97	92.04
applu	2.82	106.65	2037.61	2076.06	78.67
apsi	3.6	54.89	183.44	1224.86	79.56
fpppp	3.33	103.62	774.13	1837.96	134.62
hydro2d	3.09	144.80	147.67	242.08	52.14
mgrid	3.34	1876.11	3933.03	4003.44	286.48
su2cor	3.22	38.21	34.81	55.56	47.60
swim	3.10	112.08	112.08	275.21	89.15
tomcatv	3.61	32.85	61.47	119.67	58.91
turb3d	3.42	370.98	482.24	3652.46	0
wave5	3.25	29.28	35.71	35.71	0

What is Parallelism?

- Independent units of work that can execute concurrently if sufficient resources exist

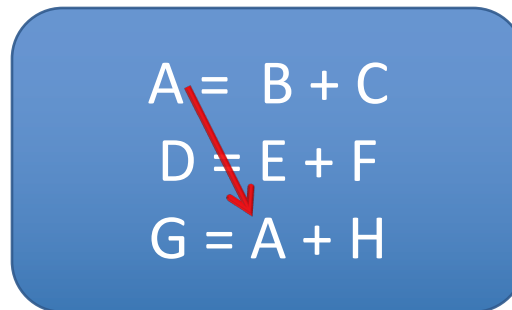


Where to find parallelism?

- Parallelism can be found/exists at different granularities
 - Instruction Level
 - Data Level
 - Thread Level
 - Task Level

Instruction-Level Parallelism (ILP)

- A measure of how many operations in a sequential program can be performed simultaneously



- Micro-architectural and compiler techniques are employed to extract ILP
 - Instruction pipelining, superscalar execution, out-of-order execution, VLIW, dataflow architecture

Data Level Parallelism (DLP)

- DLP is parallelism inherent in program loops where similar operation sequences are performed on elements of a large data structure
- Need compiler and programmer's help in extracting DLP

```
for (i=0; i<N; i++)  
    A[i] = C x B[i];
```

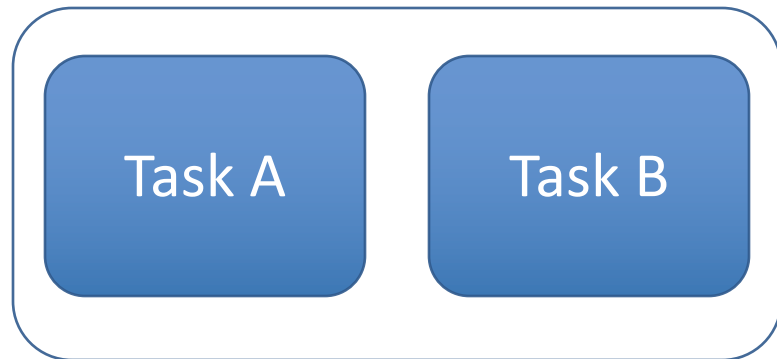
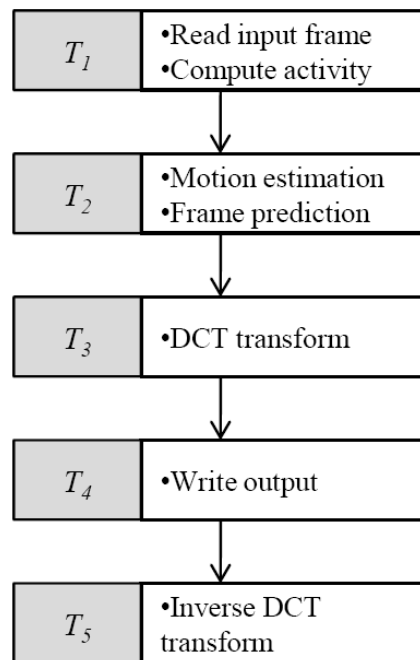
Thread Level Parallelism (TLP)

- Higher level of parallelism available as multiple threads of control within a process
- Need compiler and programmer's help in extracting TLP

```
for (i=0; i<200; i++)  
    for (j=1; j<20000; j++)  
        val [i,j] = val[i,j-1] +1;
```

Task Level Parallelism

- Higher level of parallelism where different processes execute on the same or different data
- Need user and programmer's help in indentifying task parallelism



Learning Objectives

- Factors that impact processor performance
 - Definition of CPI (clock per instr) and IPC (instr per clock)
- What is instruction-level parallelism (ILP)
 - Dependences and Hazards
- Simple pipeline (static scheduling)
- Dynamic scheduling and out-of-order execution
 - Scoreboard
 - Tomasulo's algorithm
 - Tomasulo + Speculation (Branch Prediction)

Processor Performance Equations (1/2)

$$CPU\ time = CPU\ clock\ cycles \times clock\ period$$

$$CPU\ time = \frac{CPU\ clock\ cycles}{clock\ frequency}$$

$$CPI(Clocks\ per\ Instruction) = \frac{CPU\ clock\ cycles}{Instruction\ count}$$

$$IPC\ (Instructions\ per\ clock) = \frac{1}{CPI}$$

$$CPU\ time = Instruction\ count \times CPI \times clock\ period$$

Processor Performance Equations (2/2)

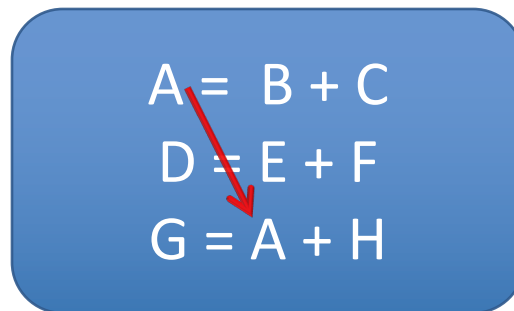
$$CPU \text{ clock cycles} = \sum_{i=1}^n IC_i \times CPI_i$$

$$CPI = \frac{\sum_{i=1}^n IC_i \times CPI_i}{IC} = \sum_{i=1}^n \frac{IC_i}{IC} \times CPI_i$$

IC is the instruction count

Instruction-Level Parallelism (ILP)

- A measure of how many operations in a sequential program can be performed simultaneously



- Micro-architectural and compiler techniques are employed to extract ILP
 - Instruction pipelining, superscalar execution, out-of-order execution, VLIW, dataflow architecture

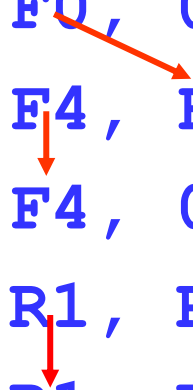
Dependences among instructions

- If two instructions are dependent, they are not parallel and must be executed in order --- although they may often be partially overlapped
- Data dependence or True data dependence
- Name dependence
- Control dependence

Data dependence (1/2)

- Instr j is data dependent on instruction i if
 - Instr i produces a result that may be used by j
- OR
- Instr j is data dependent on instr k and instr k is data dependent on instr i

```
Loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #8
        BNE     R1, R2, LOOP
```



Data dependence (2/2)

- Data dependences are properties of program
- Data value may flow between instructions through registers or memory
- Executing dependent instructions simultaneously causes the pipeline to detect the hazard (if pipeline depth is longer than the distance between the instructions in cycles) and stall
- Overcoming data dependences
 - Maintaining the dependence but avoiding the hazard --- scheduling
 - Eliminating a dependence through code transformation

Name dependence

- Two instructions use the same register or memory location, called a name, but no flow of data between the instructions through that name
- Name dependence between an instr i that precedes instr j in program order
 - Antidependence: instr j writes a register or memory that instr i reads
 - Output dependence: instr i and instr j write the same register or memory location
- Instructions can execute in parallel if the name is changed so that the instructions do not conflict --- register renaming

Data Hazards (1/2)

- A data hazard is created whenever there is dependence (data dependence or name dependence) between instructions and they are close enough during execution that could change the order of access of the operands involved in the dependence
- Instr i precedes instr j in program order
 - RAW (read after write): true data dependence; j tries to read a source before i writes it
 - WAW (write after write): output dependence; j tries to write an operand before i writes it
 - WAR (write after read): antidependence; j tries to write a destination before it is read by i

Data Hazards (2/2)

- RAW is the most common type of hazard in processor pipeline and limits ILP
- WAW is present only in pipelines that writes in more than one pipe stages or allow an instr to proceed even when a previous instr is stalled
- WAR cannot occur in most static issue pipelines as all reads are early (ID) and all writes are late (WB). Occurs when instructions are reordered

Control Dependence

- Determines the ordering of an instruction i with respect to a branch instruction

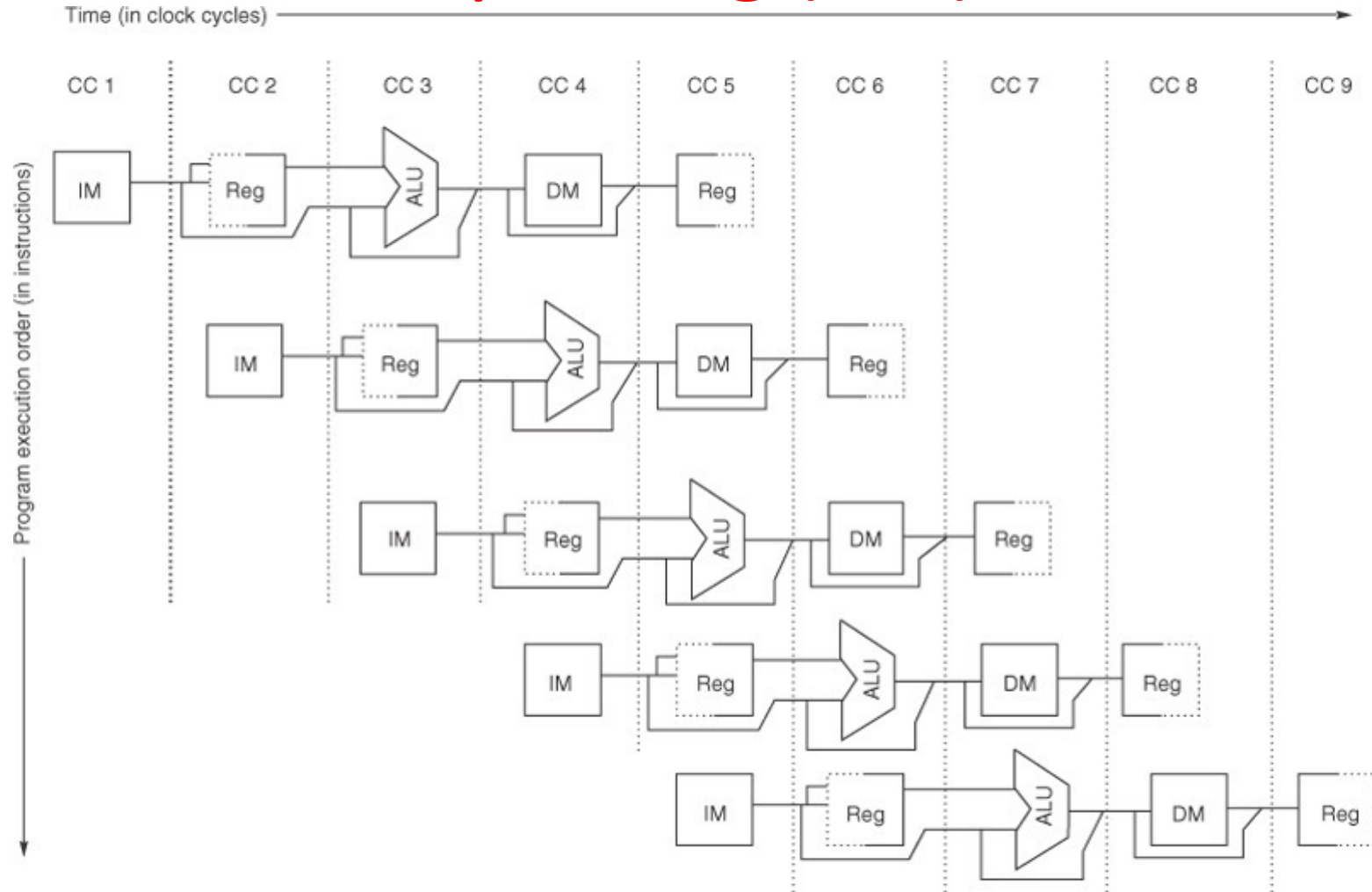
`if p1 {S1;}` S1 is control dependent on p1

- An instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
- An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Processor Pipelining (1/2)

- A technique to exploit **instruction-level parallelism (ILP)** by overlapping execution of multiple instructions
- Instruction execution is divided into multiple pipeline stages
- 5 stages in our example: IF, ID, EX, MEM, WB
- **Start a new instruction in each clock cycle**
- Different pipeline stages are completing different parts of different instructions in parallel

Processor Pipelining (2/2)



Pipelining Lessons

- Each pipeline stage corresponds to a **processor cycle**
- The **length of a processor cycle is the time required for the slowest pipeline stage** --- usually 1 clock cycle
- Use processor cycle and clock cycle interchangeably
- Each instruction takes 5 clock cycles to execute
- Pipelining does not improve **latency** of a single instruction
- Pipelining helps **throughput** --- number of instructions executed in unit time

Ideal Pipeline

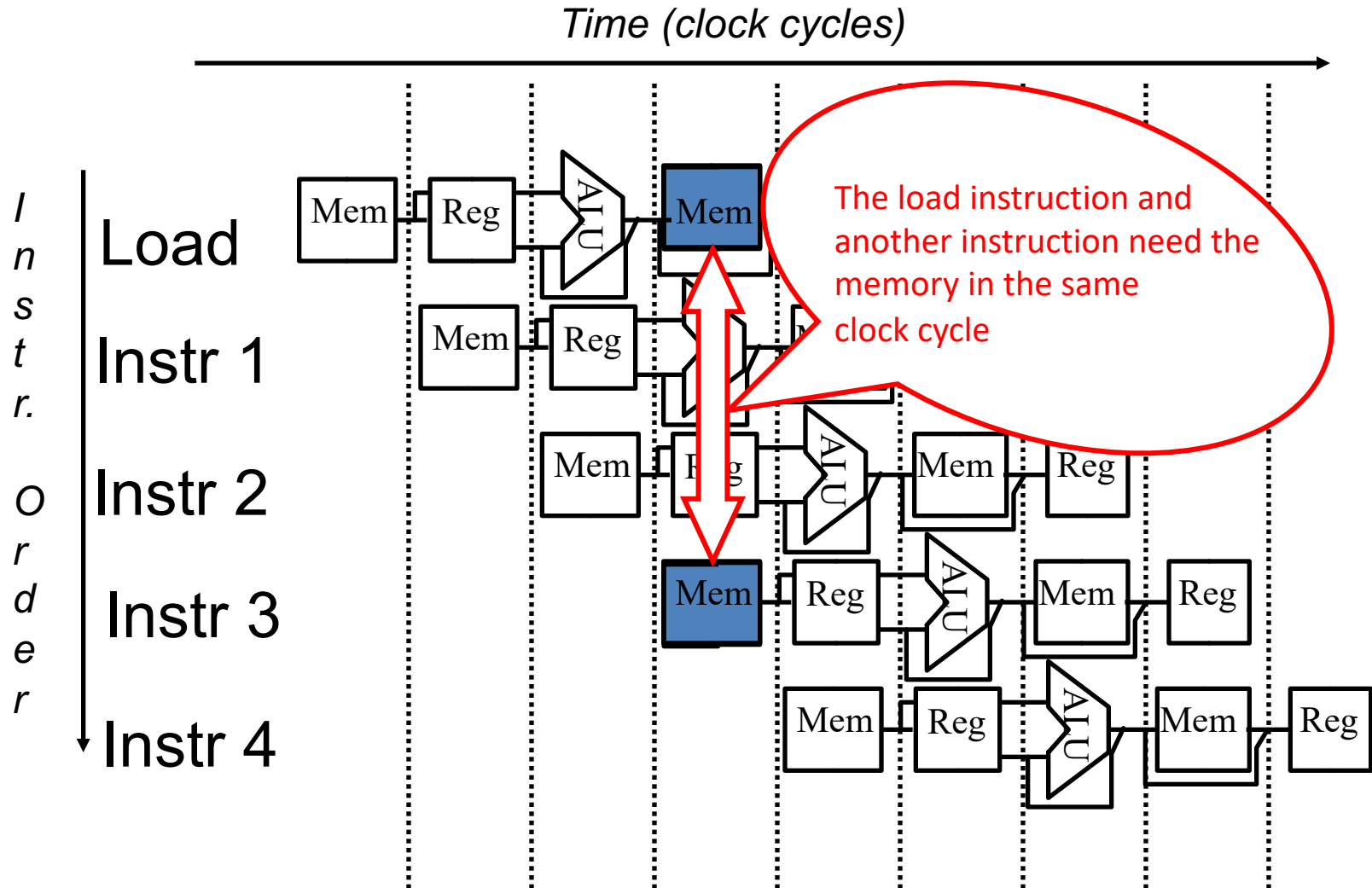
- Perfectly balanced pipeline stages
- There are no stalls for dependencies
- In the steady state, one instruction completes execution every cycle
- Time to “fill up” the pipeline is negligible if number of instructions executed is large
- CPI for ideal pipeline =
(clock cycles/instruction count) = 1
- Assuming 1ns clock period, average instruction execution time = 1ns
- Speedup is equal to number of pipeline stages

Pipeline Hazards

- Trouble for pipeline
- Hazards can prevent next instruction from immediately following previous instruction
 - pipeline stalls
- Structural hazards:
 - Simultaneous use of a hardware resource
- Data hazards:
 - Data dependencies between instructions
- Control hazards:
 - Change in program flow

Example of Structural Hazard

Only one memory



Time (clock cycles)



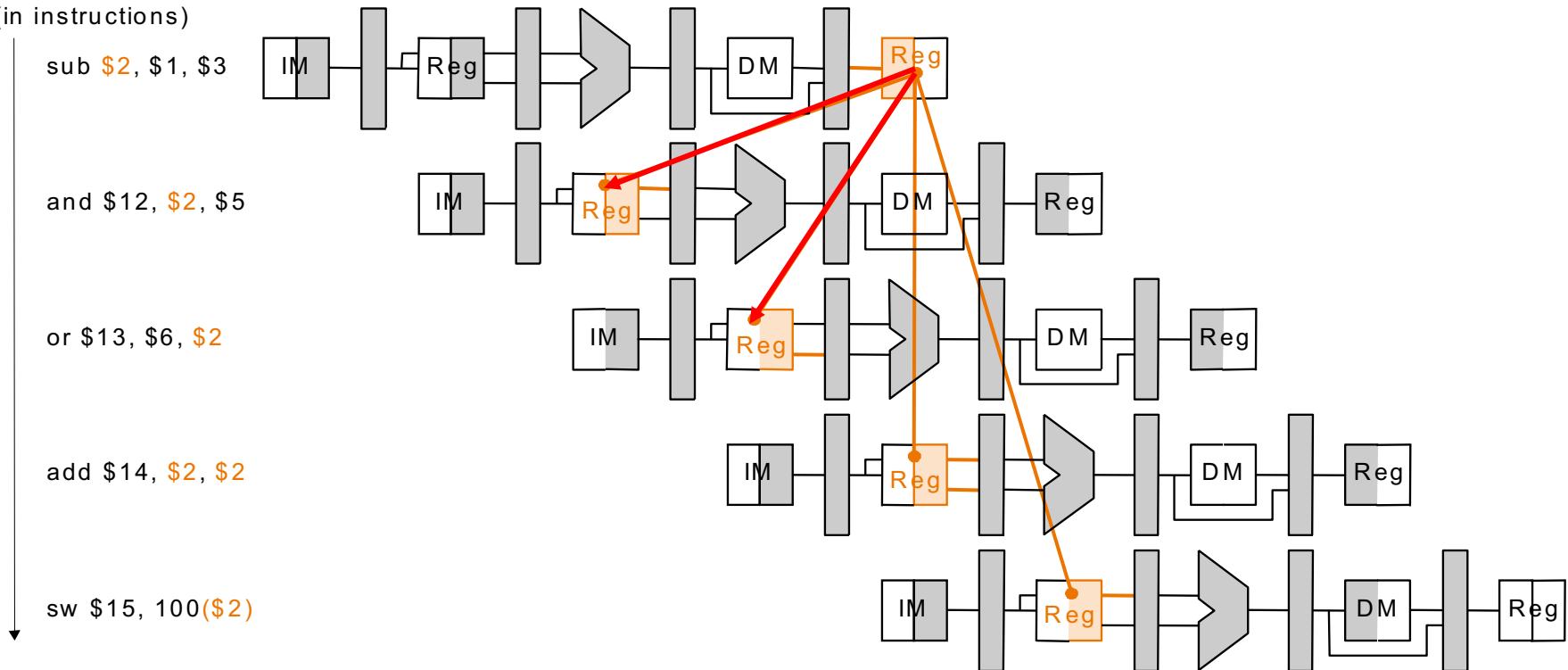
Data Hazards: Example

- Value from prior instruction is needed before write back

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register \$2:	10	10	10	10	10/- 20	- 20	- 20	- 20	- 20

Program
execution
order
(in instructions)



30

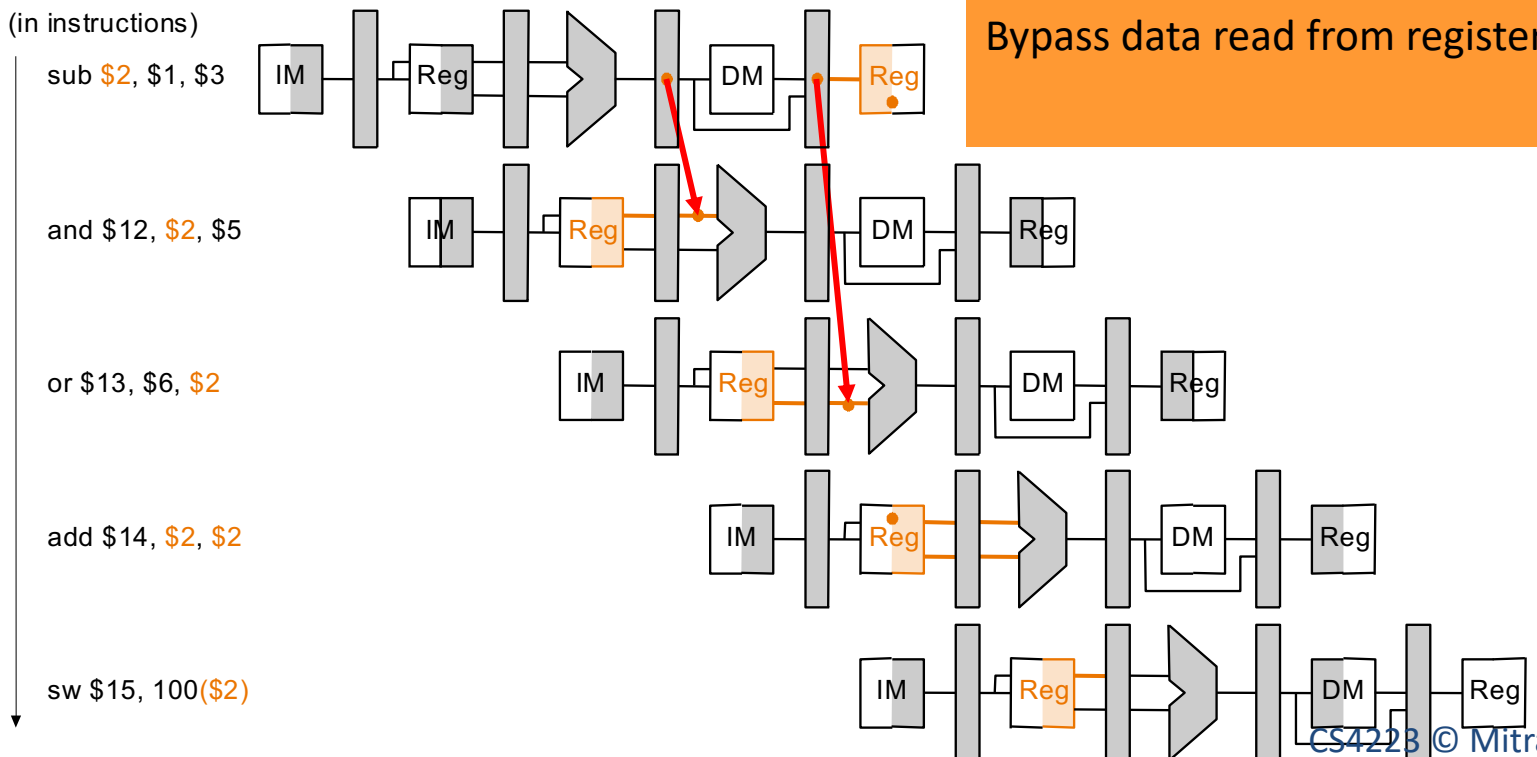
Observations

- When is the data from **sub** instruction actually produced?
 - End of **EX** stage for **sub** or clock cycle 3
- When is the data actually needed by **and**?
 - Beginning of **and**'s **EX** stage or clock cycle 4
- When is the data actually needed by **or**?
 - Beginning of **or**'s **EX** stage or clock cycle 5
- Simply forward the data as soon as it is available to any units that need it before it is available to read from the register file
 - Bypass the data read from the register file

Solution: Forwarding/Bypassing

	Time (in clock cycles) →								
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register \$2 :	10	10	10	10	10/-20	-20	-20	-20	-20
Value of EX/MEM :	X	X	X	-20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	-20	X	X	X	X

Program
execution order
(in instructions)



Forward results from one stage to another
Bypass data read from register file

Control Hazards

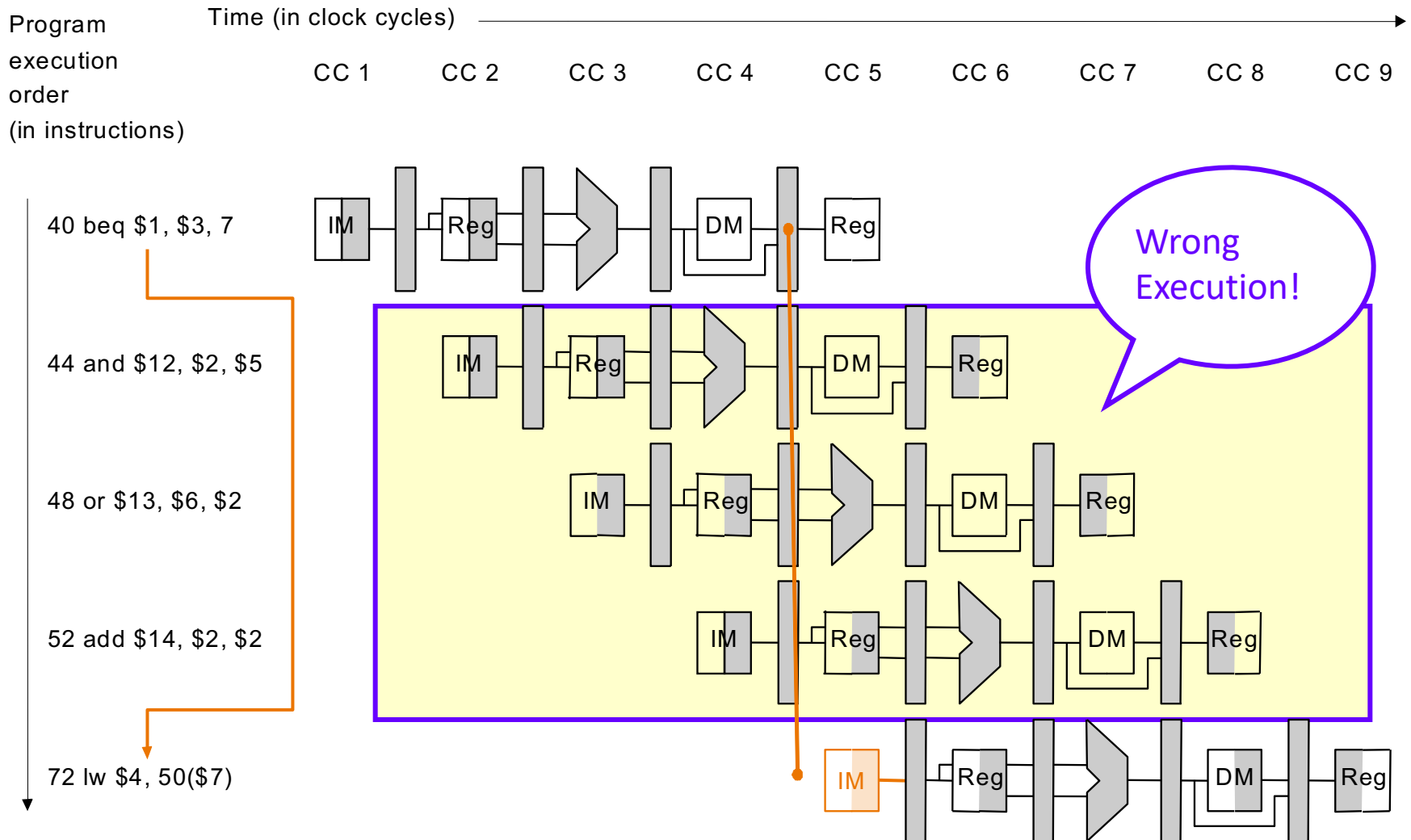
- Make decision about which instruction to execute next based on result of an instruction
- Instruction sequence

$\$1 \neq \3

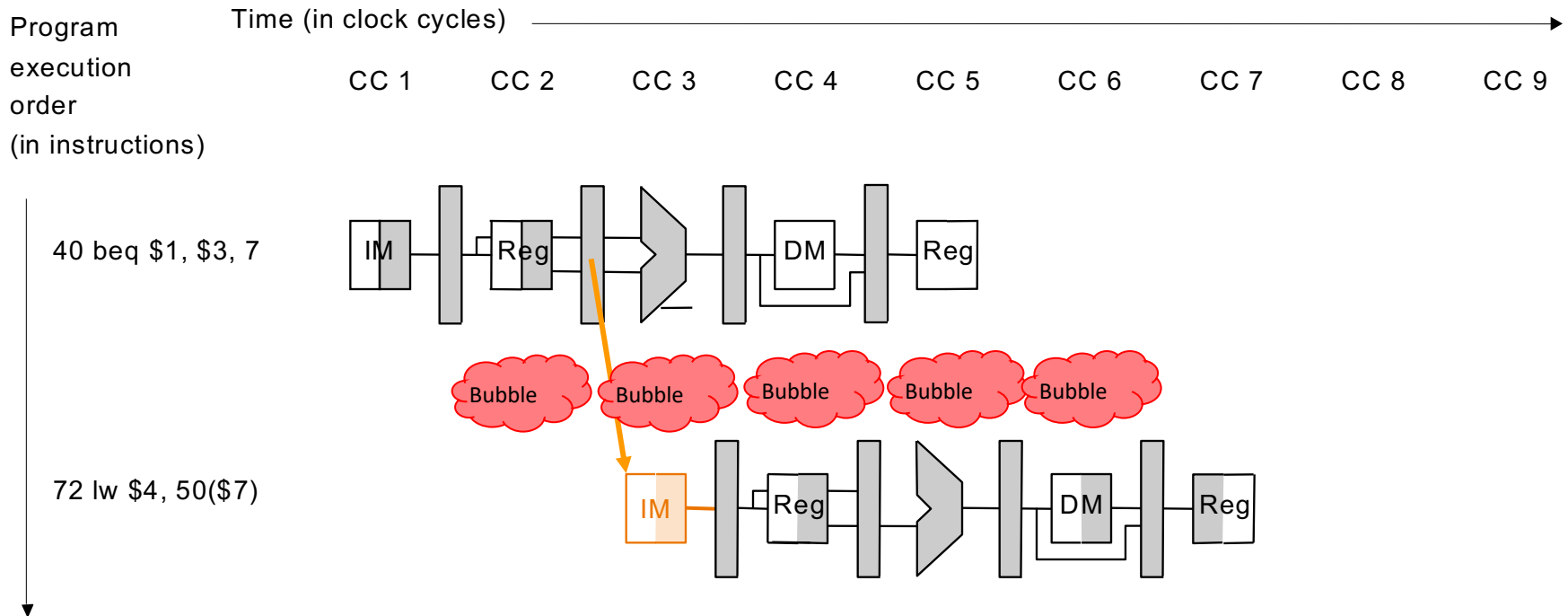
```
40  beq  $1, $3, 7
44  and  $12, $2, $5
48  or   $13, $6, $2
52  add  $14, $2, $2
.....
72  lw   $4, 5($7)
```

$\$1 = \3

Control Hazards: Example

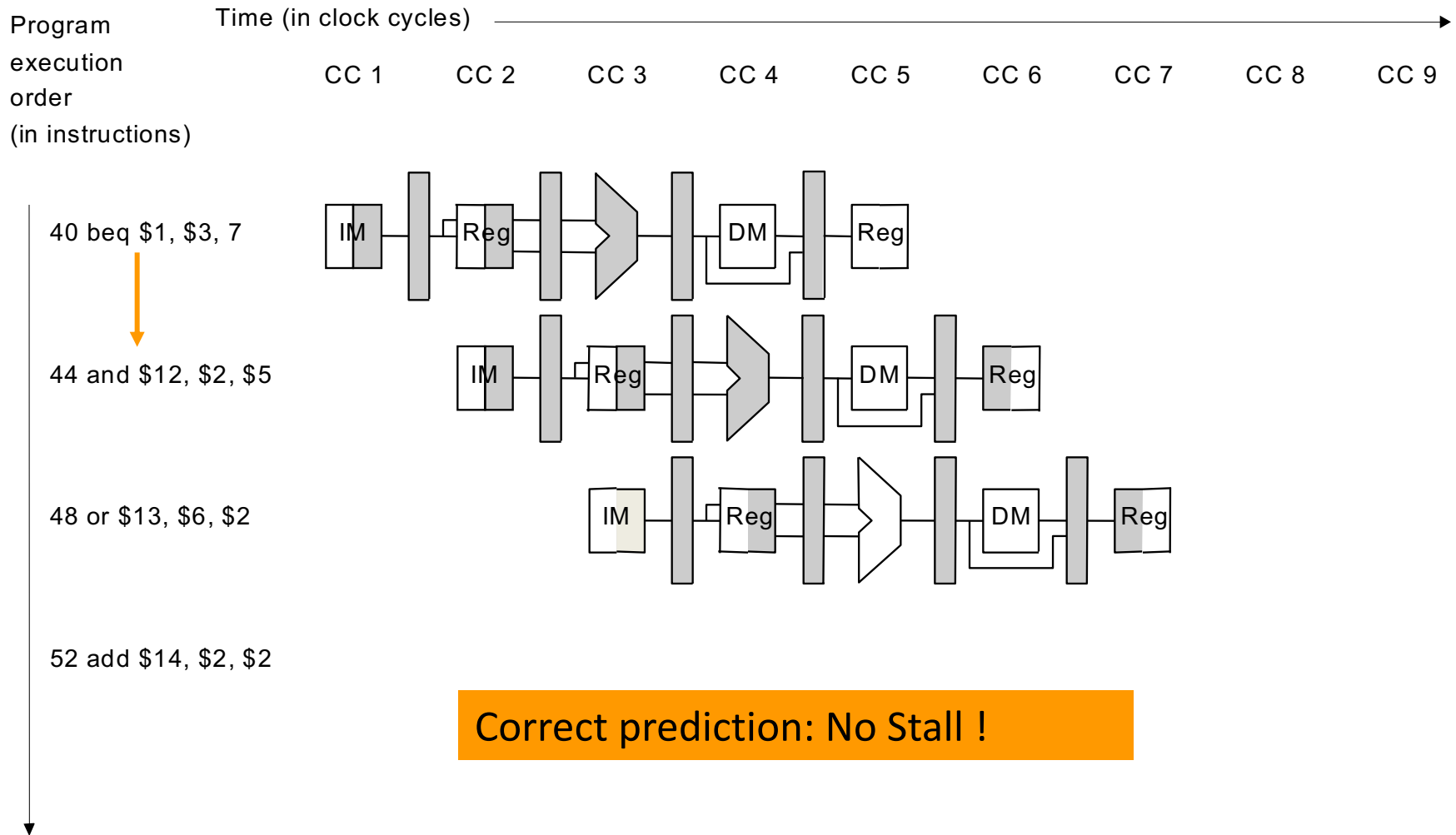


Control Hazards: Reduce stalls

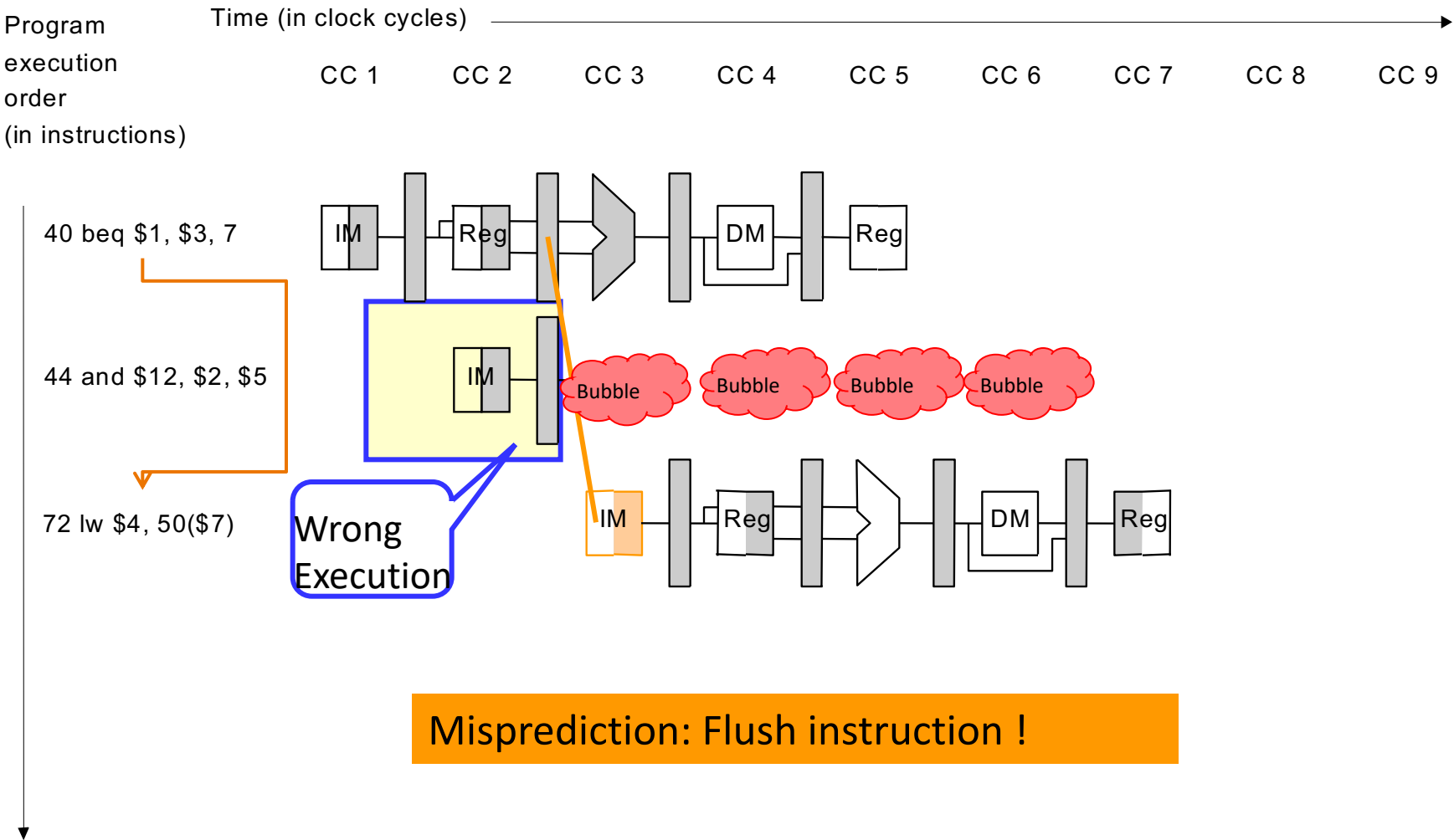


- Move decision making to ID stage
- Wait till you know the branch decision and then fetch the correct instructions
- Introduces 1 clock cycle delay!

Control Hazards: Branch prediction



Control Hazards: Branch prediction



Pipeline so far..

- Simple statically scheduled
- Fetches, issues, and executes instructions in **program order**
- Stalls when true data dependency cannot be hidden with forwarding or there is a control dependency
- Stalls all the instructions behind as well
- Cannot exploit instruction level parallelism (ILP) to the fullest

Exploiting ILP

- Compile-time approach depends on compiler to find parallelism statically
 - Limited use in scientific or DSP code
 - Example Intel Itanium
- Dynamic scheduling depends on hardware to find parallelism at runtime
 - Dominates the market, Intel Pentium, AMD, ARM

Dynamic Scheduling

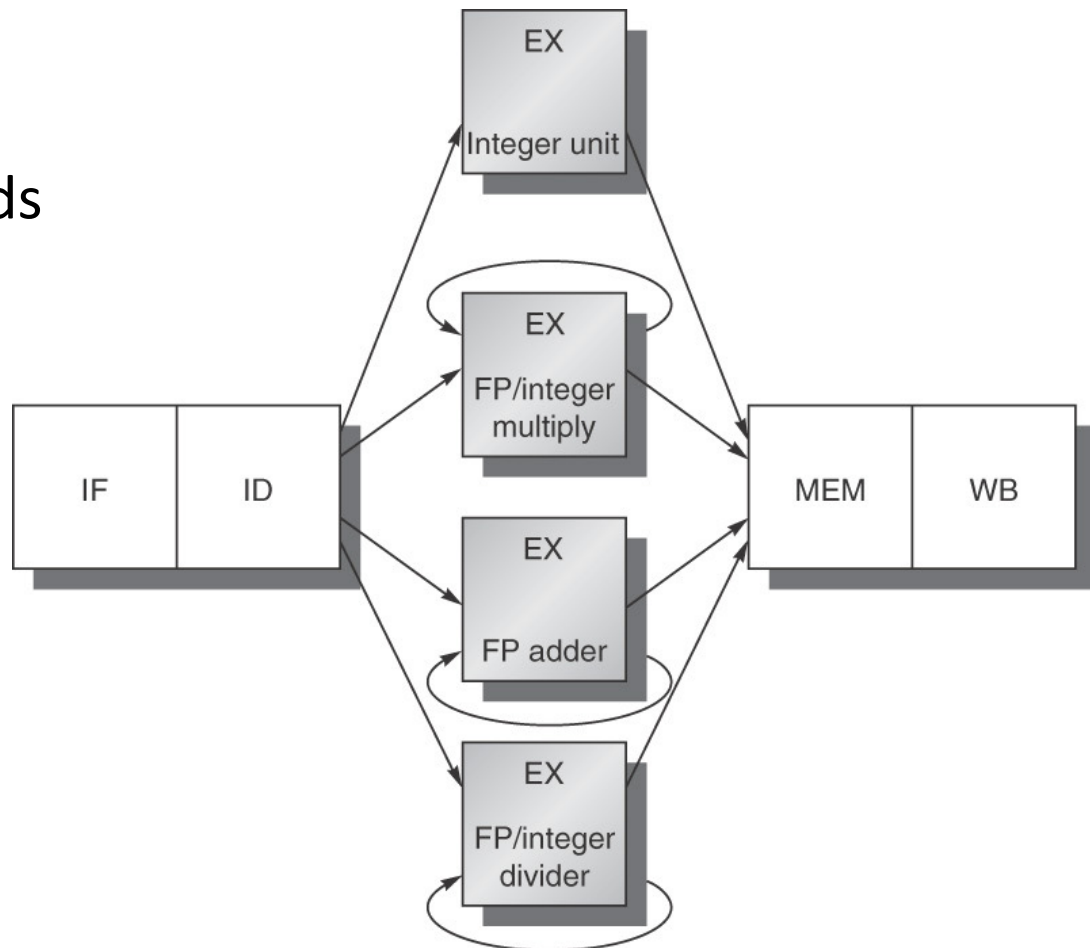
- Hardware rearranges instruction execution to reduce stalls while maintaining data flow
- Goal: Instruction-Level Parallelism (ILP) should be limited only by true data dependencies
 - Preserve program order only where it affects outcome of the program
 - Rename registers to overcome name dependences

Why Dynamic Scheduling?

- Simple MIPS pipeline requires every instruction to complete EX stage in 1 cycle
- Impractical for floating-point operations
 - Slower clock
 - Enormous logic for floating point units
- Allow floating-point pipeline to have longer and varying latency of operations
 - EX stage may take multiple cycles
 - There may be multiple floating-point FUs
 - Main integer unit (integer ALU, load/store, branch address)
 - FP and integer multiplier
 - FP adder
 - FP and integer divider

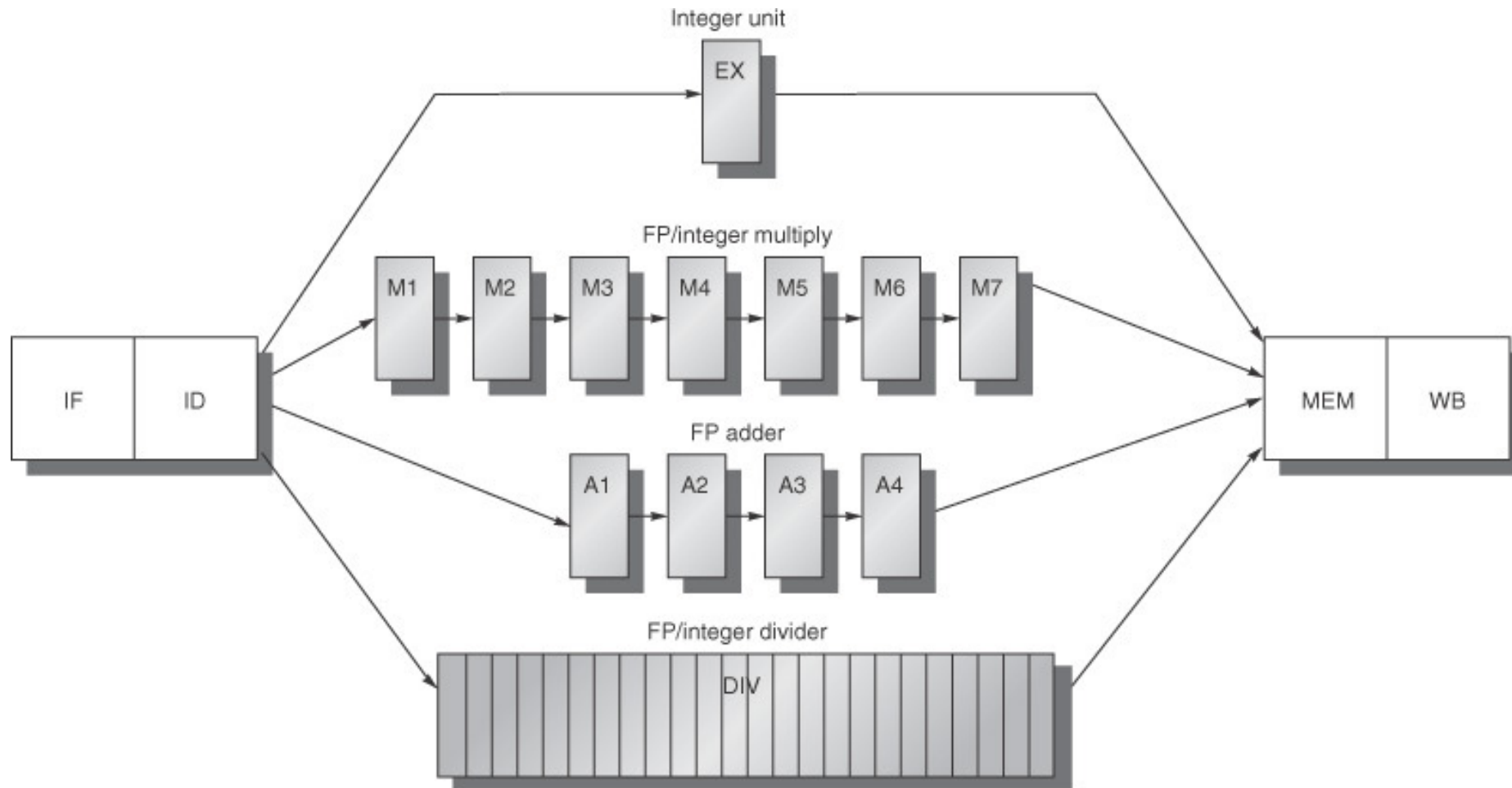
Unpipelined Floating-point FUs

Frequent
Structural hazards



© 2007 Elsevier, Inc. All rights reserved.

Pipelined Floating-point FUs



© 2007 Elsevier, Inc. All rights reserved.

Latency and Initiation Interval

- **Latency:** Number of intervening cycles between an instr that produces a result and an instr that uses the result
- **Initiation Interval:** Number of cycles that must elapse between issuing two operations of a given type
- Pipeline latency is essentially equal to 1 cycle less than the depth of the execution pipeline (first to last EX stage) except
 - Load execution pipeline: 1 cycle later
- Latency to a store is 1 cycle less for data to be stored
- Faster clock rate → more pipeline stage

Hazards

- Structural hazards for FP divider
- Number of register writes in a cycle can be larger than 1 as instructions have varying running times
- WAW hazards are possible as instructions no longer reach WB in order
 - WAR hazards are not possible
- Longer latency of operations implies more frequent stalls due to RAW hazards

RAW hazards

	Clock cycle Number																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4, 0(R2)	IF	ID	EX	MEM	WB												
MUL.D F0, F4, F6		IF	ID		M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2, F0, F8			IF		ID							A1	A2	A3	A4	MEM	WB
S.D F2, 0(R2)					IF							ID	EX				MEM

Structural hazards

	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
		IF	ID	EX	MEM	WB					
			IF	ID	EX	MEM	WB				
ADD.D F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
					IF	ID	EX	MEM	WB		
						IF	ID	EX	MEM	WB	
L.D F2, 0(R2)							IF	ID	EX	MEM	WB

WAW hazards

	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
		IF	ID	EX	MEM	WB					
			IF	ID	EX	MEM	WB				
ADD.D F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
					IF	ID	EX	MEM	WB		
L.D F2, 0(R2)						IF	ID	EX	MEM	WB	

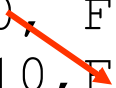
Dynamic Scheduling

- Hardware rearranges the instruction execution to reduce the stalls while maintaining data flow
- Handles cases when dependences unknown at compile time
 - it allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve
- Allows code that is compiled for one pipeline to run efficiently on a different pipeline
- Simplifies the compiler
- **Hardware speculation** a technique with significant performance advantages, builds on dynamic scheduling

Basic Idea

- Allow instructions behind stall to proceed

```
DIV.D    F0, F2, F4  
ADD.D    F10, F0, F8  
SUB.D    F12, F8, F14
```



- Split ID stage into two parts
 - **Issue:** Decode instructions, check for structural hazards
 - **Read operands:** Wait until no data hazards, then read operands
- Enables **out-of-order execution** and **out-of-order completion**
- Still **in-order instruction issue**

Out-of-order Execution

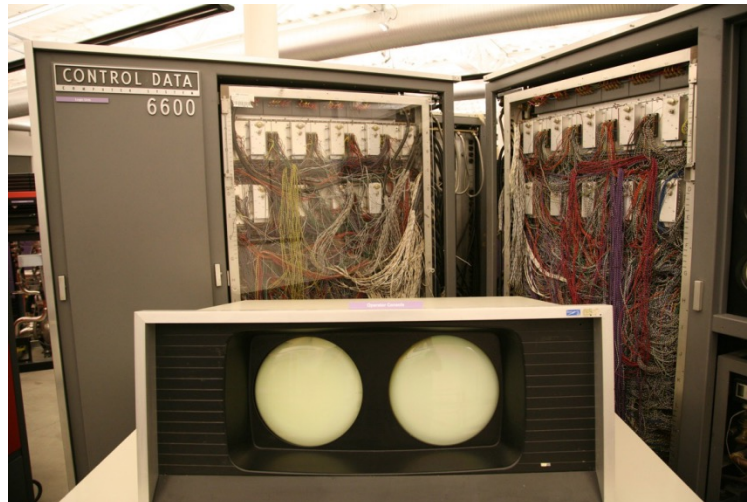
- Distinguish when an instr **begins execution** and when it **completes execution**; between the two times the instruction is **in execution**
- Introduces possibility of both WAR and WAW hazards

```
DIV.D    F0, F2, F4
ADD.D    F6, F0, F8
SUB.D    F8, F10, F14
MUL.D    F6, F10, F8
```

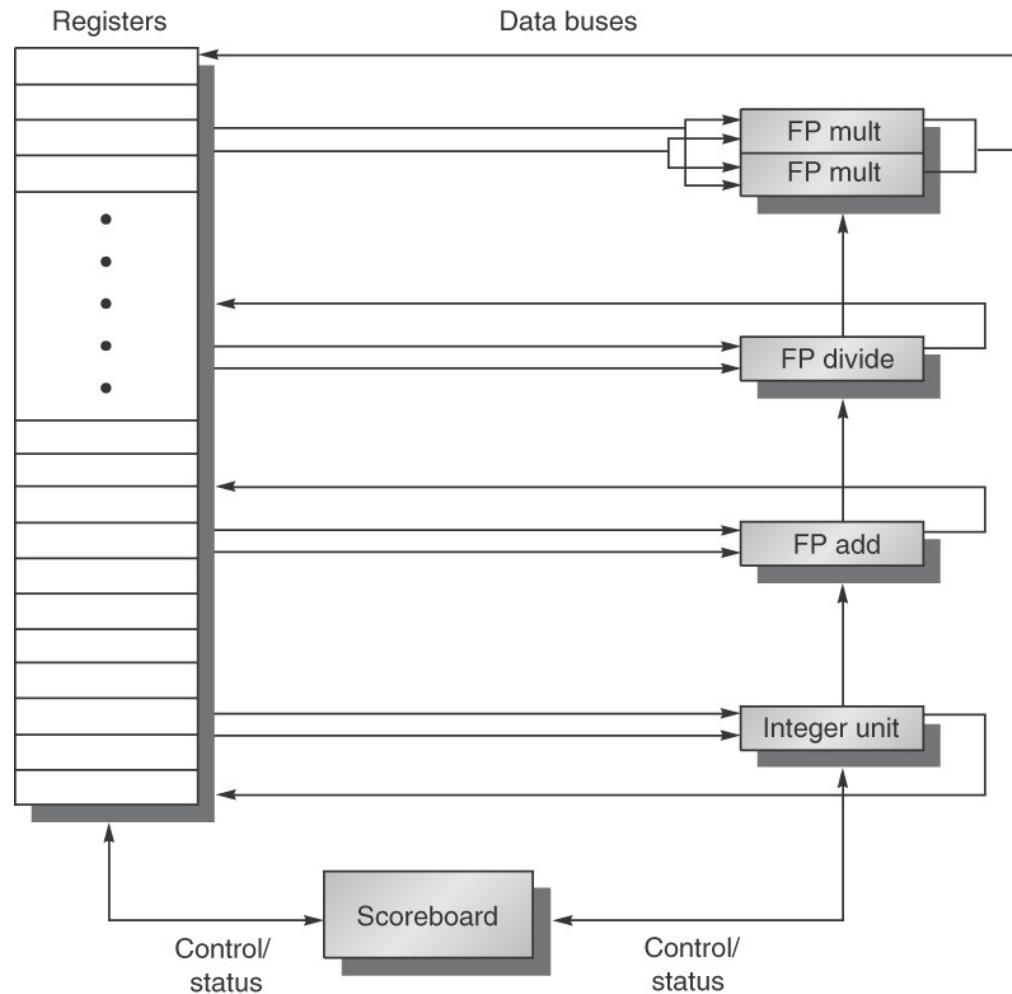
The diagram illustrates out-of-order execution with four instructions. Red arrows show data dependencies: the first instruction (DIV.D) writes to F0, which is read by the second (ADD.D); the third instruction (SUB.D) writes to F8, which is read by the second (ADD.D); and the fourth instruction (MUL.D) writes to F6, which is read by the second (ADD.D). This shows that the second instruction depends on the completion of the first, third, and fourth instructions, even though it is executed second.

Dynamic Scheduling with Scoreboard

- Allows instructions to execute out-of-order when there are sufficient resources and no data dependences
- Named after CDC 6600 scoreboard
- Hazard detection and resolution are centralized in scoreboard



Scoreboard



© 2007 Elsevier, Inc. All rights reserved.

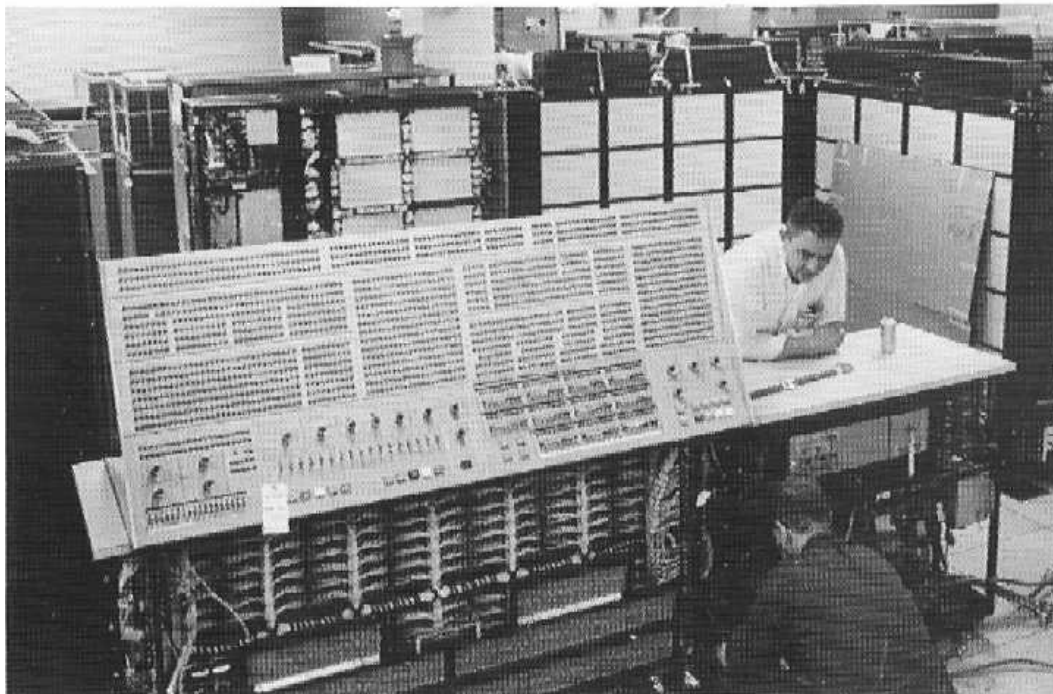
Pipeline stages with Scoreboard

- **Issue:** If FU is free and there is **no WAW hazard**, scoreboard issues the instruction to FU
- **Read operands:** When no earlier issued active instruction is going to write to the source register (**no RAW hazard**), scoreboard allows FU to read operands and begin execution (no bypassing or forwarding)
- **Execution:** When result is ready in FU, scoreboard is informed
- **Write results:** Scoreboard checks for **WAR hazard** and stalls the completed instruction, if necessary, and proceeds only when WAR hazard clears

Tomasulo's Approach

- For IBM 360/91 (before caches!)
 - ⇒ Long memory latency
- Goal: High Performance without special compilers
- Small number of floating point registers (4 in 360) prevented interesting compiler scheduling of operations
 - This led Tomasulo to try to figure out how to get more effective registers — [register renaming in hardware!](#)
- Why Study 1966 Computer?
- The descendants of this have flourished!
 - Alpha 21264, Pentium 4, AMD Opteron, Power 5, ...

IBM 360/91

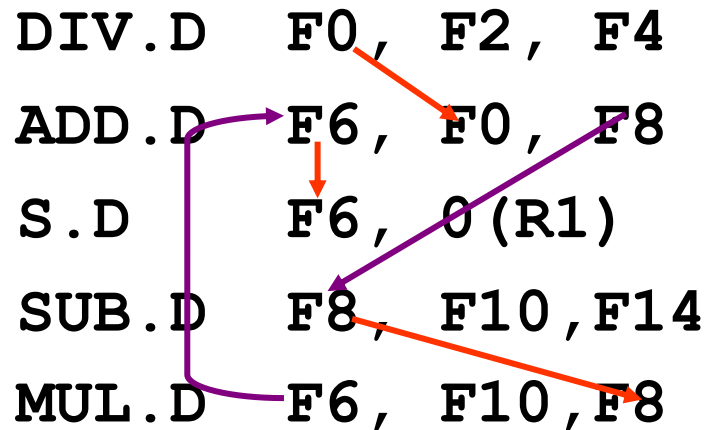


Scoreboard versus Tomasulo

- Scoreboard drawbacks
 - Instruction issue stalls in the presence of WAW hazard
 - Instruction completion (i.e., write to the register) stalls in the presence of WAR hazard
- Tomasulo's algorithm overcomes this through hardware register renaming
- Tomasulo's algorithm can be extended to support hardware speculation

Register Renaming

DIV.D F0, F2, F4
ADD.D F6, F0, F8
S.D F6, 0(R1)
SUB.D F8, F10, F14
MUL.D F6, F10, F8

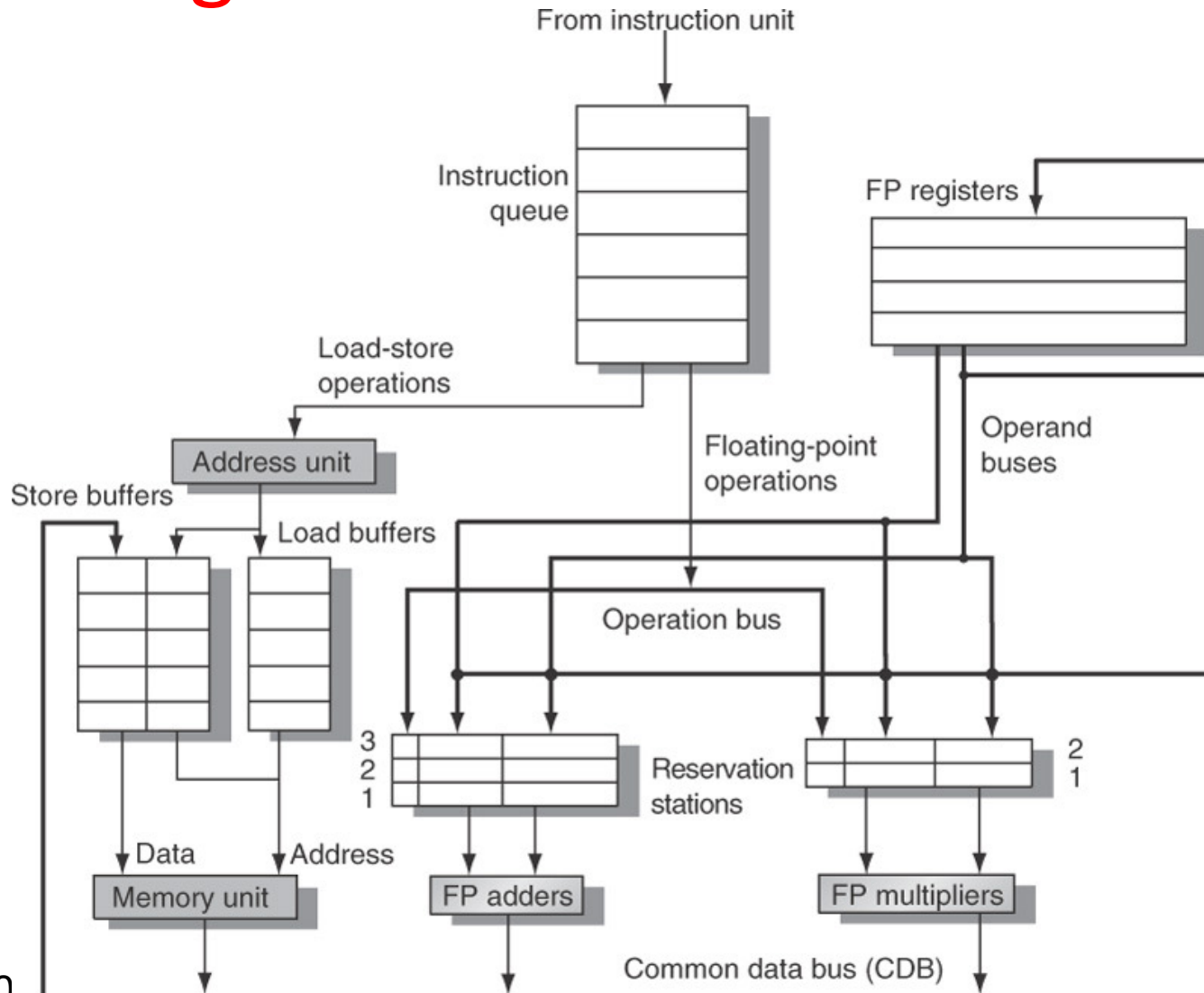


DIV.D F0, F2, F4
ADD.D S, F0, F8
S.D S, 0(R1)
SUB.D T, F10, F14
MUL.D F6, F10, T

Tomasulo's Algorithm

- Control & buffers distributed with Function Units (FU)
 - FU buffers called “reservation stations”; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations (RS); called register renaming ;
 - Renaming avoids WAR, WAW hazards
 - More reservation stations than registers, so can do optimizations compilers can't
- Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs
 - Avoids RAW hazards by executing an instruction only when its operands are available
- Load and Stores treated as FU with RS as well
- Integer instructions can go past branches (predict taken), allowing FP ops beyond basic block in FP queue --- cannot execute any FP instruction before the branch is resolved

Organization



Algorithm Steps

- **Issue:** get next instruction from instruction queue in FIFO order
 - Matching RS empty: issue instr to RS with operand values if they are in registers; otherwise keep track of instr (RS) that will produce the result
 - Matching RS busy: structural hazard; stall
- **Execute:** If operands not available, monitor CDB; when all operands available start execution
- **Write Result:** Write result on CDB and from there into registers and any RS + store buffers waiting for this result

Data Hazard through Memory

- If a load and a store access the same address, then either
 - Load before store: WAR hazard
 - Store before load: RAW hazard
 - Store before store: WAW hazard
- Effective address computation in order
- Load: check address conflict by comparing it with addresses of all active store buffers
- Store: check address conflict by comparing it with addresses of all active load and store buffers
- If conflict, don't send the instruction to load/store buffer

Tomasulo's Algorithm: Advantages

- Distribution of the hazard detection logic
 - Distributed reservation stations and the CDB
 - If multiple instructions waiting on single result, & each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB
 - If a centralized register file were used, the units would have to read their results from the registers when register buses are available
- Elimination of stalls for WAW and WAR hazards

Tomasulo's Algorithm: Drawbacks

- Complexity
- Each reservation station must contain high-speed associative buffer
- Performance limited by Common Data Bus
 - Each CDB must go to multiple functional units
 - High capacitance, high wiring density
 - Number of functional units that can complete per cycle limited to one!
 - Multiple CDBs → more FU logic for parallel associative stores


Control Dependence Revisited

- `if p1 {S1;} if p2 {S2;}`
- Control dependence is not a critical property to be preserved
 - We may executed instructions that shouldn't have been executed, thereby violating control dependence, **if** we can do so without affecting the **correctness** of the program
- Program correctness
 - **Data flow**

Data flow

- Actual flow of data values among instructions that produce results and those that consume them
- Branches makes data flow dynamic: determines which predecessor actually deliver a data value to an instruction
- Maintaining control dependence maintains data flow

```
DADD  R1, R2, R3
BEQZ  R4, L
DSUBU R1, R5, R6
L: ...
      OR   R7, R1, R8
```



The diagram illustrates data flow between instructions. Two red arrows originate from the 'R1' operand in the 'DADD R1, R2, R3' instruction. One arrow points to the 'R1' operand in the 'DSUBU R1, R5, R6' instruction, and the other points to the 'R1' operand in the 'OR R7, R1, R8' instruction. This shows how the value produced by the DADD instruction is consumed by two subsequent instructions.

Key Question

- How can we violate control dependence, if required for performance reasons, without violating program correctness (data flow)?

Hardware Speculation

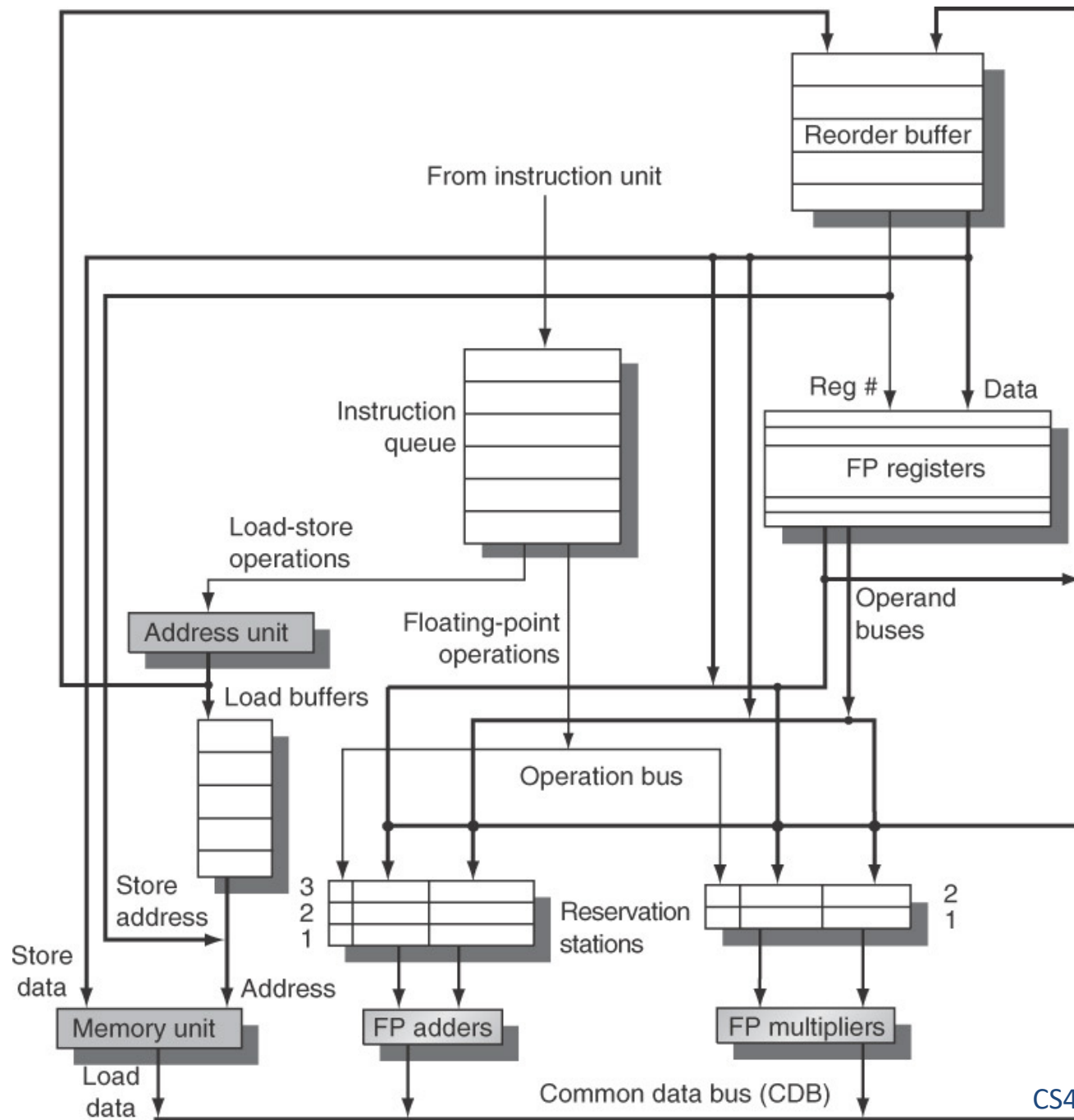
- Overcoming control dependence by speculating on the outcome of branches and executing the program as if our guesses were correct
- Three key techniques
 - **Dynamic branch prediction**
 - **Speculation**: allow the execution of instructions before the control dependences are resolved (with the ability to undo the effects of an incorrectly speculated sequence)
 - **Dynamic scheduling**: deal with the scheduling of different combinations of basic blocks

Tomasulo + Speculation

- Tomasulo: in-order issue, out-of-order execution, **out-of-order completion**
- Tomasulo + speculation: in-order issue, out-of-order execution, **in-order completion**
- Key idea: Separate bypassing of results among instructions (speculative register read) from the actual completion of an instruction (performing updates of register and memory that cannot be undone) until we know the instr is no longer speculative

Instruction Commit

- Updating the register file or memory when an instruction is no longer speculative
- Reorder Buffer (ROB)
 - Additional hardware buffers to hold the results of instructions that have finished execution but have not committed
- ROB supplies operands in the interval between completion of instruction execution and instruction commit
- ROB acts like additional registers
- Renaming function of the reservation stations is replaced by ROB



Reorder Buffer operation

- Holds instructions in FIFO order, exactly as issued
- When instructions complete, results placed into ROB
 - Supplies operands to other instruction between execution complete & commit \Rightarrow more registers like RS
 - Tag results with ROB buffer number instead of reservation station
- Instructions **commit** \Rightarrow values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches

Instruction Execution

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**

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 reservation station available.

4. Commit—update register with reorder result

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

Avoiding Memory Hazards

- WAW and WAR hazards through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending
- RAW hazards through memory are maintained by two restrictions:
 - not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and
 - maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.
- These restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data

Advanced Pipelining: Multiple Issue

- What's EPIC and superscalar?
- Multiple Issue processors
 - Multiple instructions in every pipeline stage
 - 4 washer, 4 dryer...
- Static multiple issue: EPIC (Explicitly Parallel Instruction Computer) or VLIW (Very Long Instruction Word)
 - Compiler specifies the set of instructions that execute together in a given clock cycle
 - Simple hardware, complex compiler
- Dynamic multiple issue: Superscalar
 - Hardware decides which instructions to execute together
 - Complex hardware, simpler compiler

Multiple-issue micro-architectures

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

ARM A7 Pipeline (Little processor)

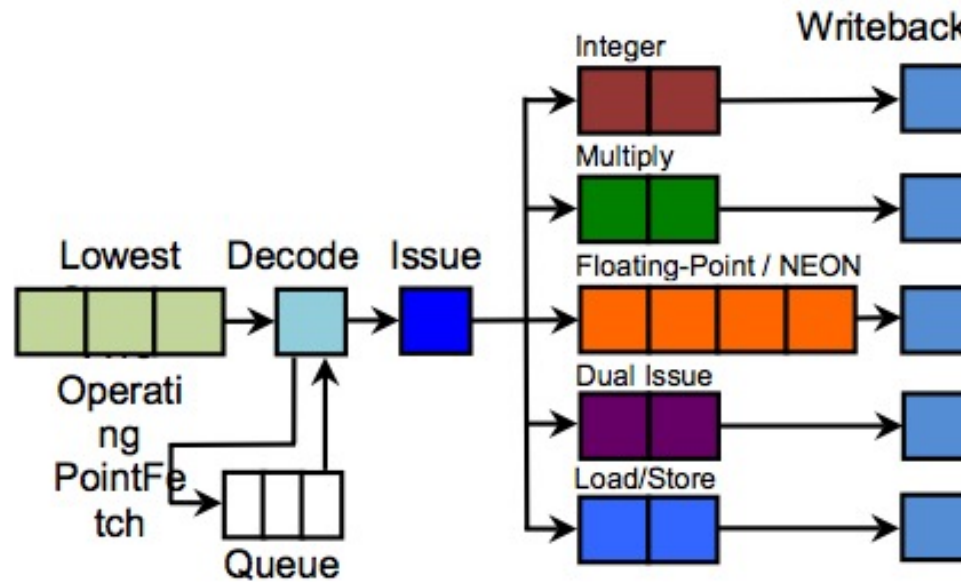
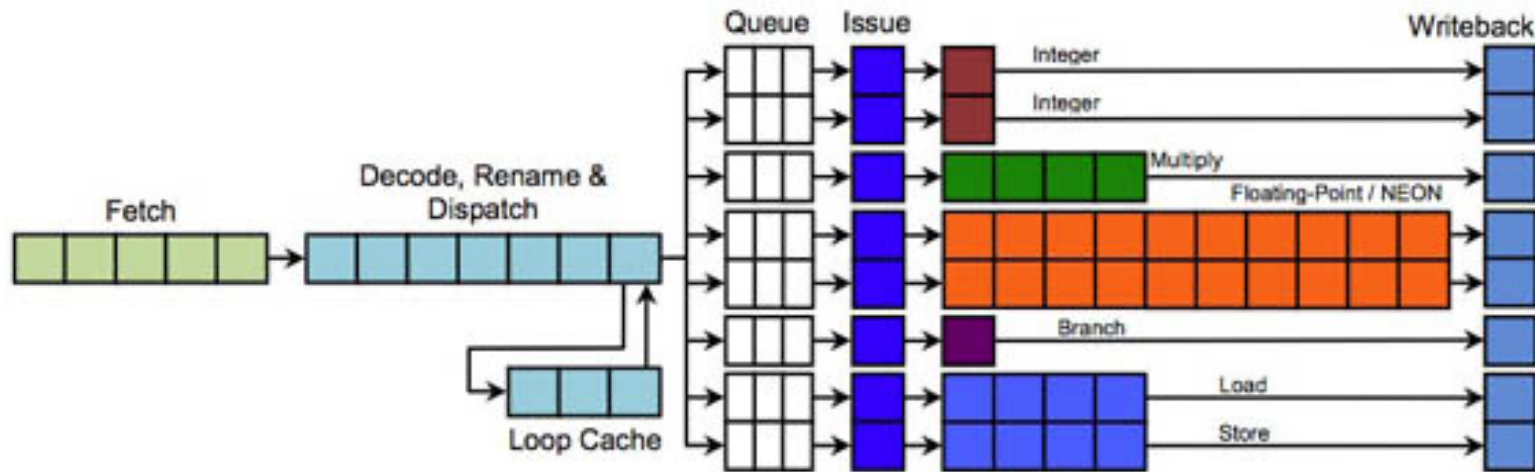
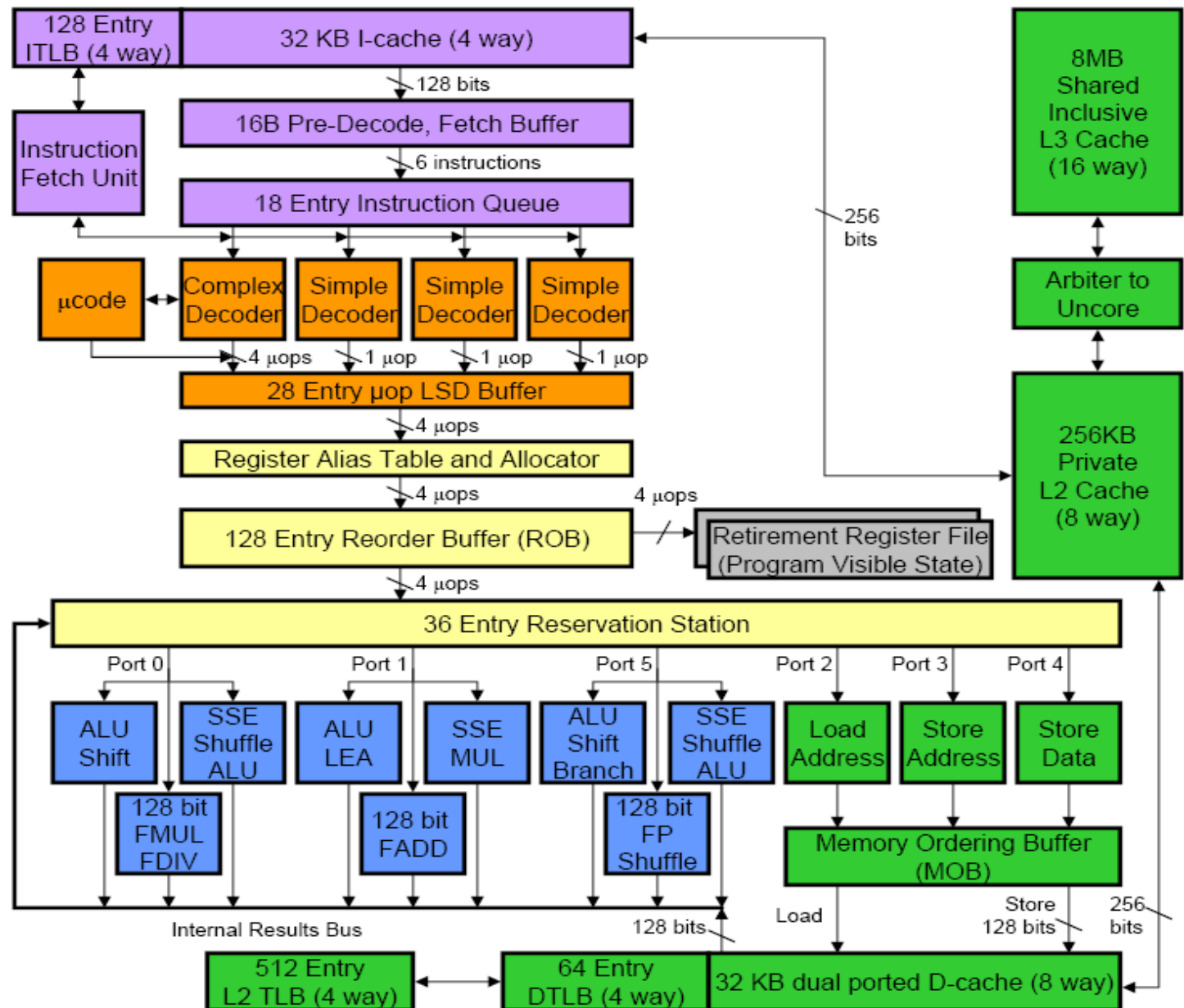


Figure 1 Cortex-A7 Pipeline

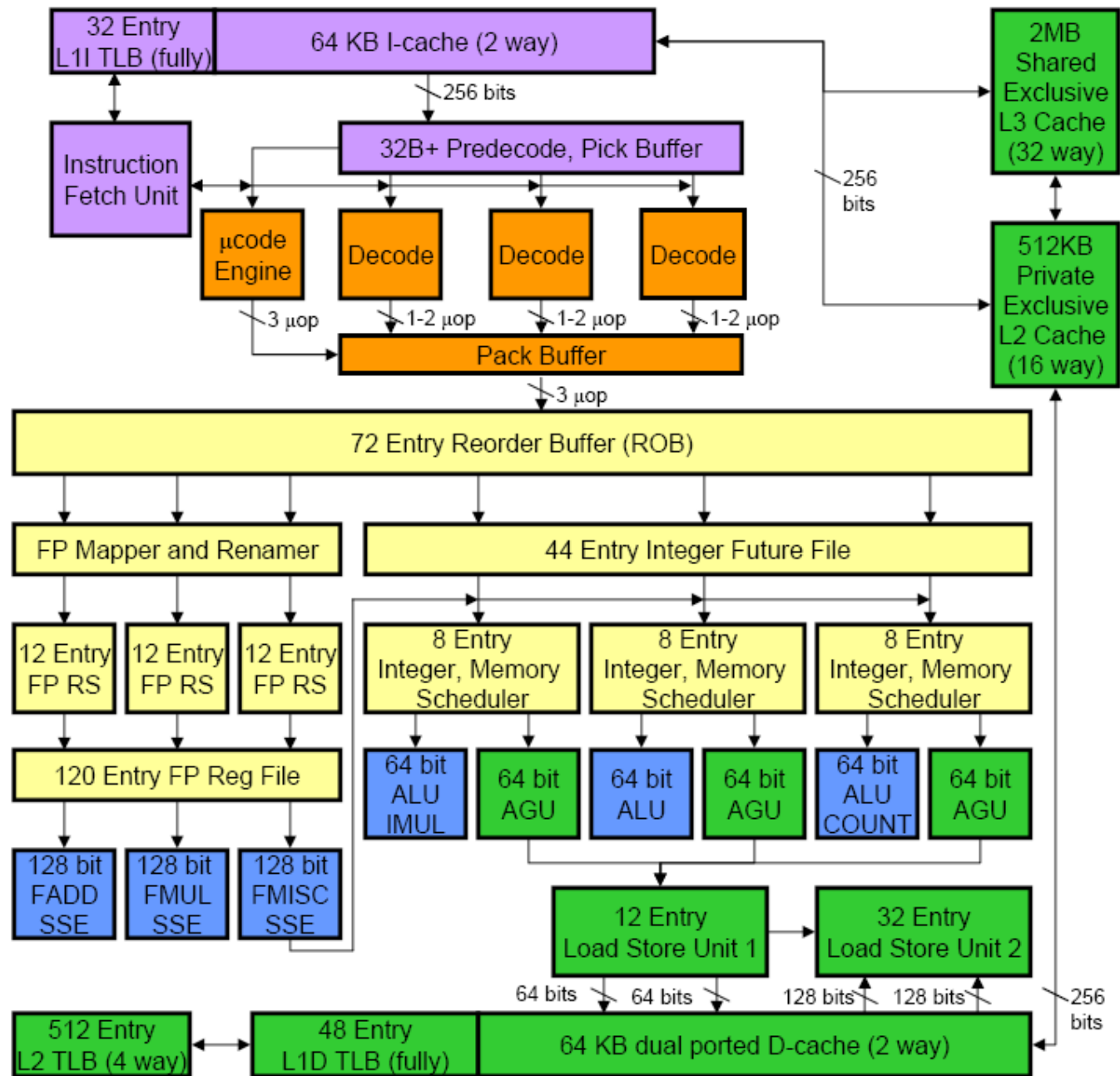
ARM A15 Pipeline (Big processor)



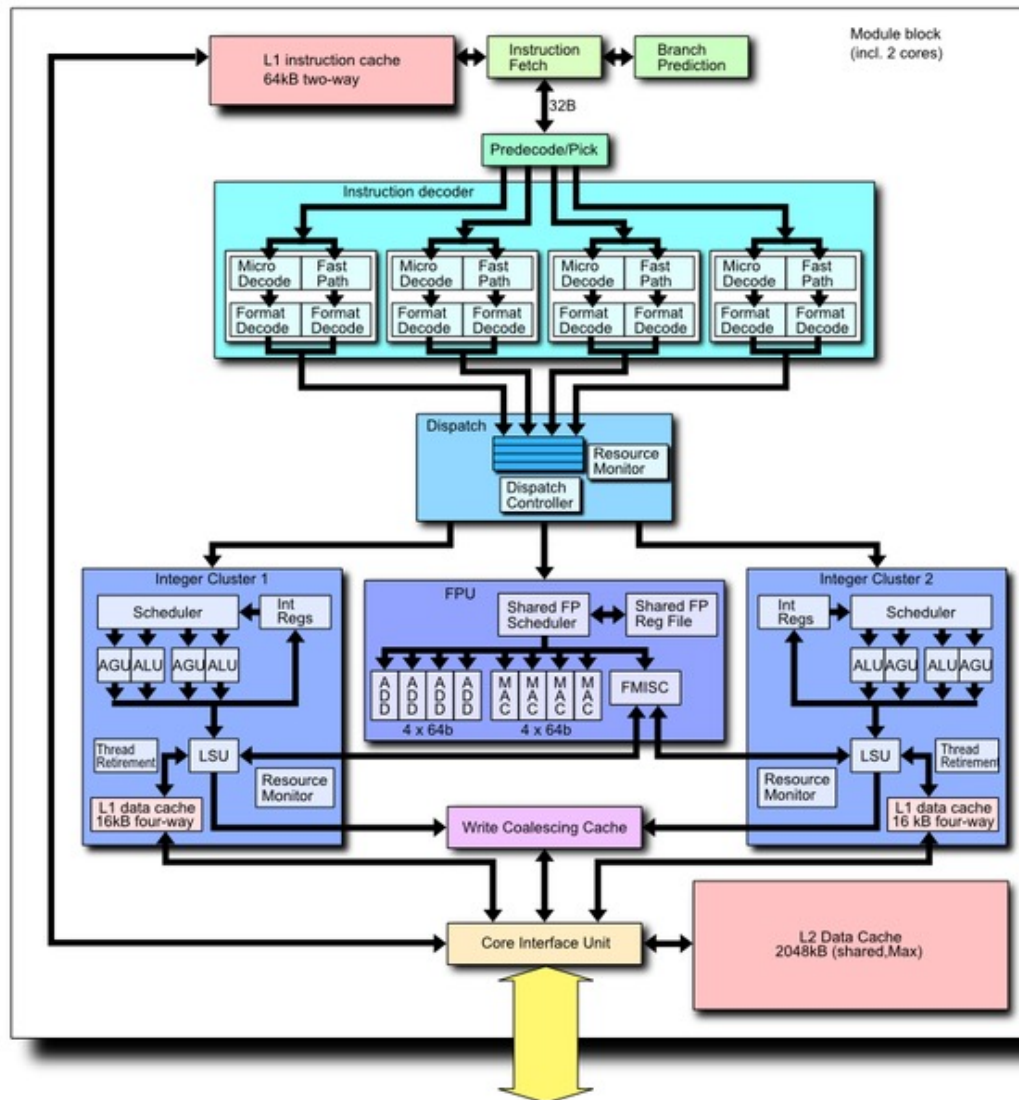
Intel Nehalem



AMD Bar- celona



AMD Bulldozer



Perspective

- Interest in multiple-issue because wanted to improve performance without affecting programming model
- Taking advantage of ILP is conceptually simple, but design problems are amazingly complex in practice
- Conservative in ideas, just faster clock and bigger cache
- Processors of last 5 years (Pentium 4, IBM Power 5, AMD Opteron) have the same basic structure and similar sustained issue rates (3 to 4 instructions per clock) as the 1st dynamically scheduled, multiple-issue processors announced in 1995
 - Clocks 10 to 20X faster, caches 4 to 8X bigger, 2 to 4X as many renaming registers, and 2X as many load-store units
⇒ performance 8 to 16X
- Peak v. delivered performance gap increasing