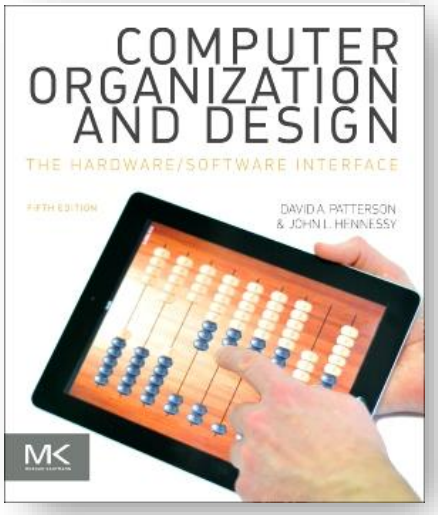
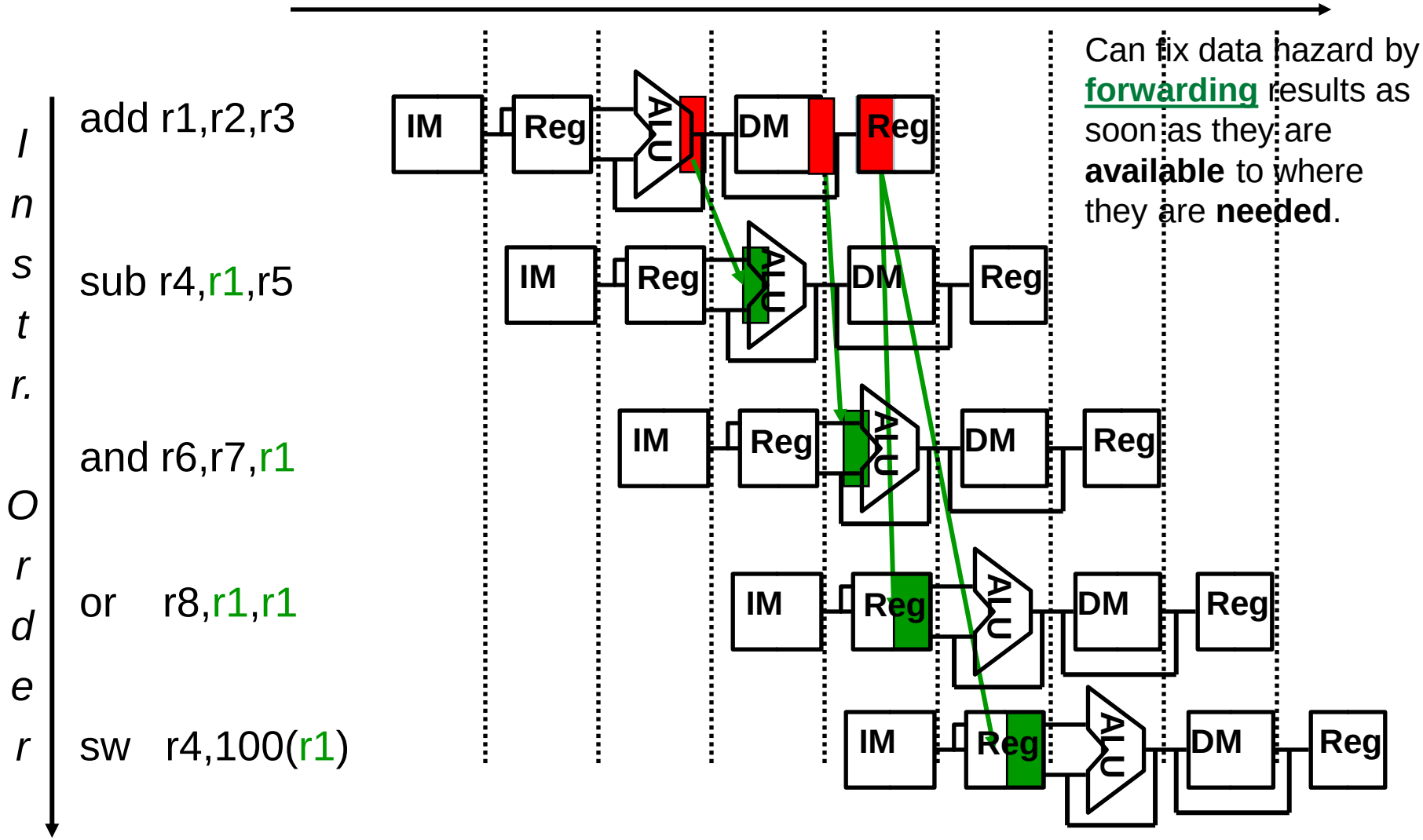


ESE 345 Computer Architecture

Pipelined Processor Control (Part 2)

Review: Another Way to "Fix" a Data Hazard



Can fix data hazard by **forwarding** results as soon as they are **available** to where they are **needed**.

Data Forwarding Control Conditions (3/4)

1. **EX/MEM hazard:**

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU provided it writes **and != R0**.

2. **MEM/WB hazard:**

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 01
```

Forwards the result from the second previous instr. to either input of the ALU provided it writes **and != R0**.

What's wrong with this hazard control?

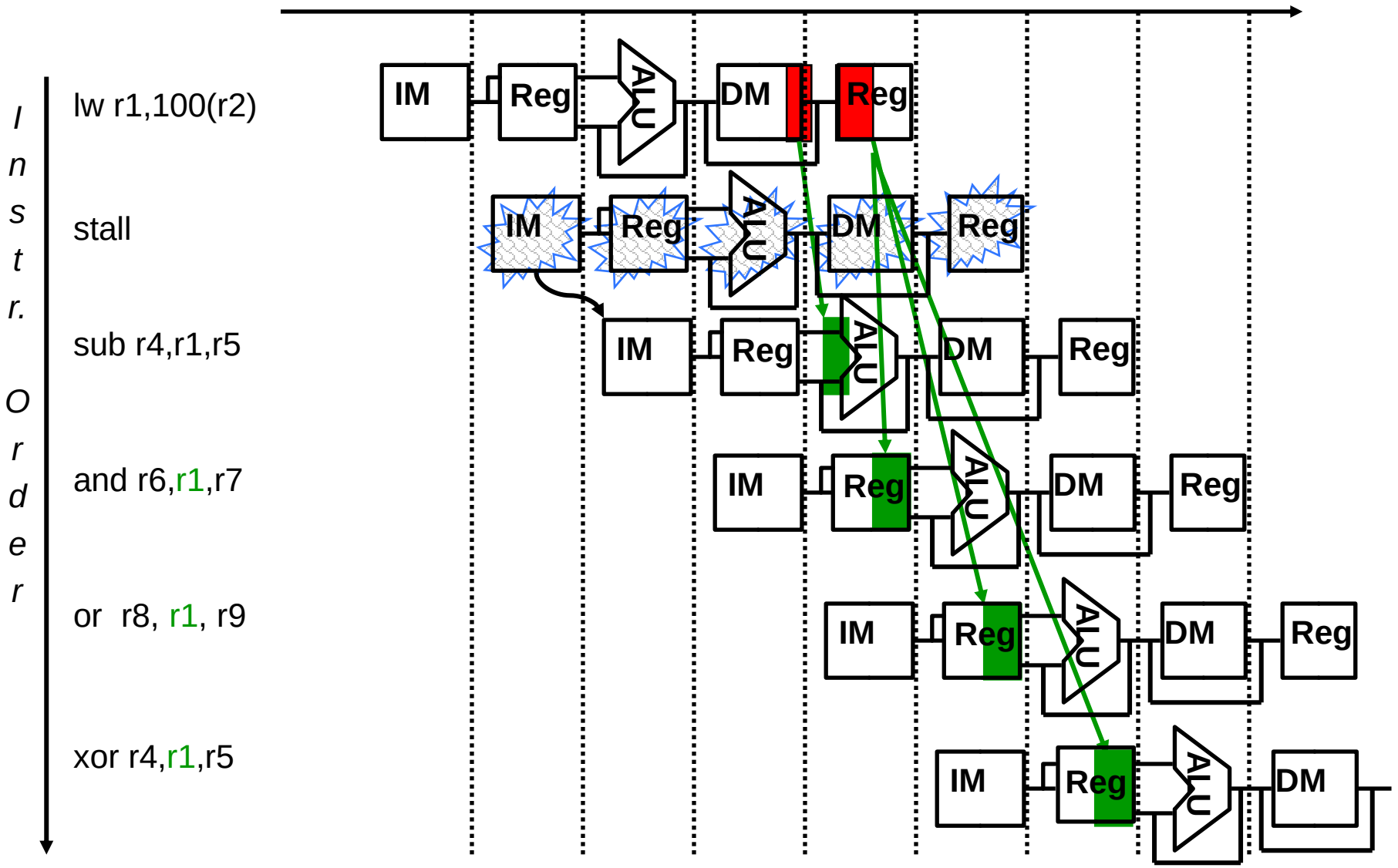
Corrected Data Forwarding Control Conditions (4/4)

2. MEM/WB hazard:

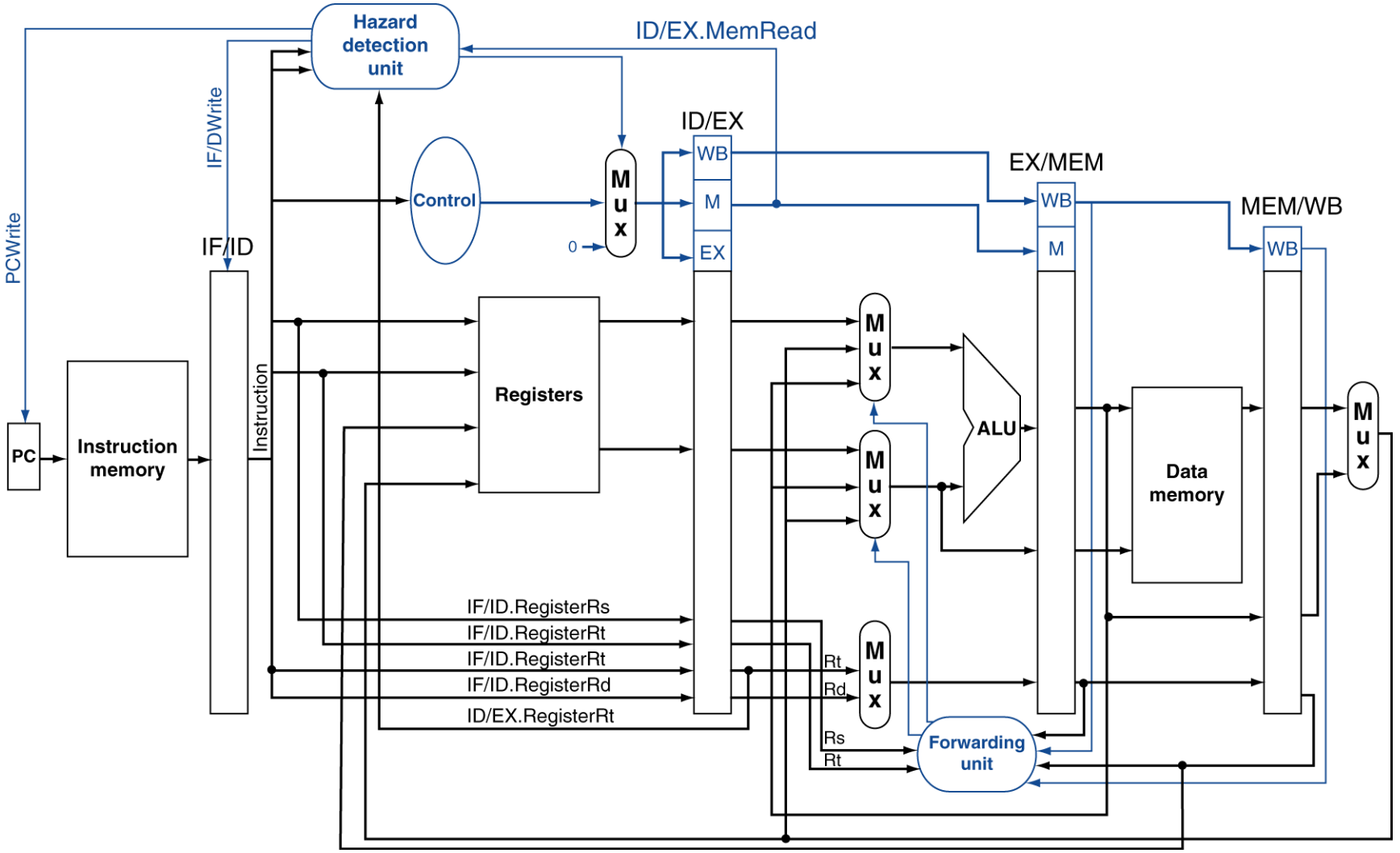
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRs)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs
   || ~ EX/MEM.RegWrite))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRt)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt
   || ~ EX/MEM.RegWrite))
      ForwardB = 01
```

Forward if this instruction writes
AND its not writing R0 AND this dest reg == source AND in between instr either dest. reg. doesn't match OR it doesn't write reg.

Forwarding with Load Data Hazards



Datapath with Hazard Detection

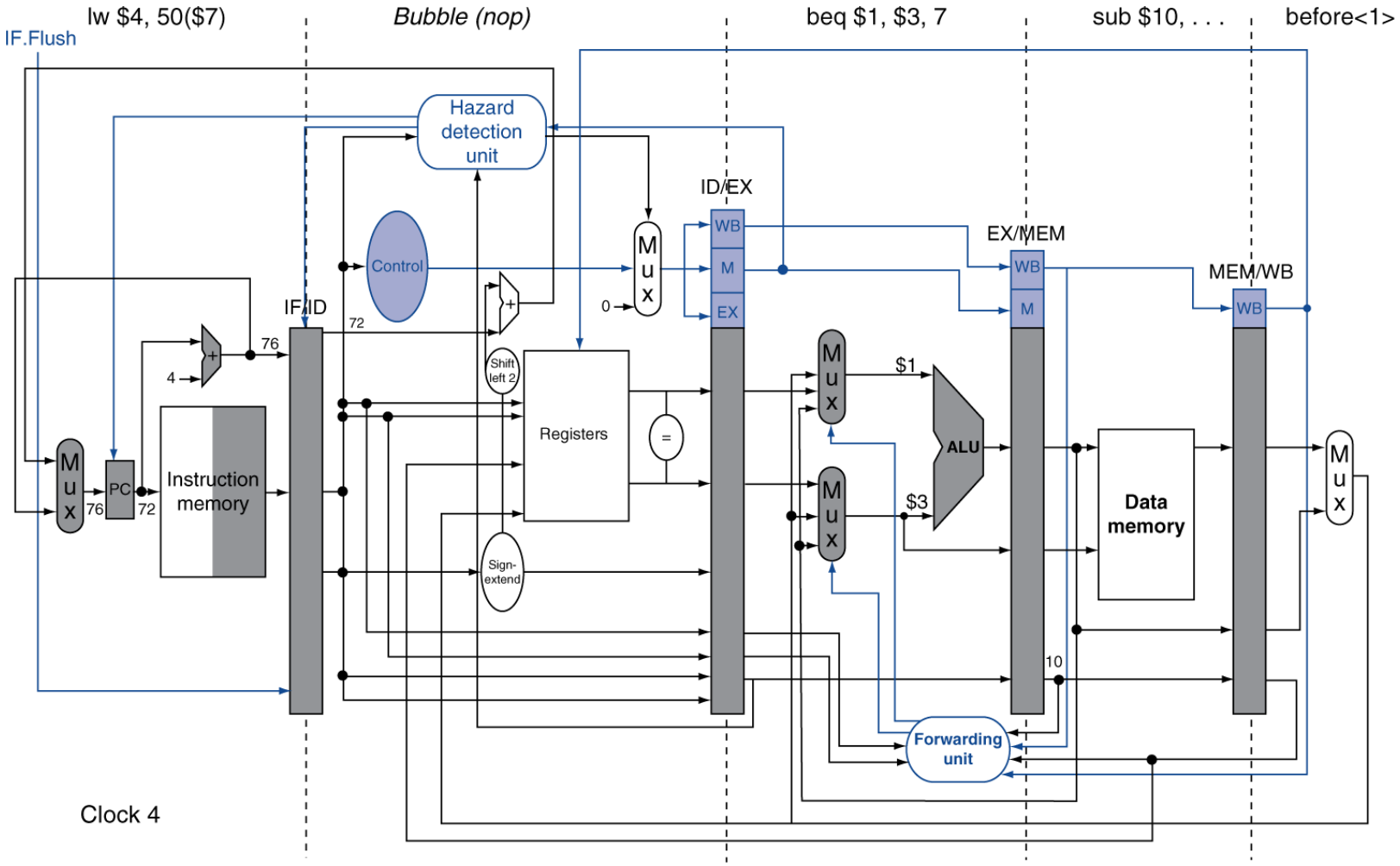


Reducing Branch Delay

- Move hardware to determine outcome from MEM to ID stage
  - Reduces the number of stall cycles to one (like with jumps)
  - We will need these two units in the ID stage:
    - Target address adder
    - Register comparator
  - For longer pipelines, decision points are later in the pipeline, incurring more stalls, so we need a better solution
- Example: branch taken

```
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7
44:   and   $12, $2, $5
48:   or    $13, $2, $6
52:   add   $14, $4, $2
56:   slt   $15, $6, $7
      ...
72:   lw    $4, 50($7)
```

Example: Branch Taken



Early Branch Forwarding Issues

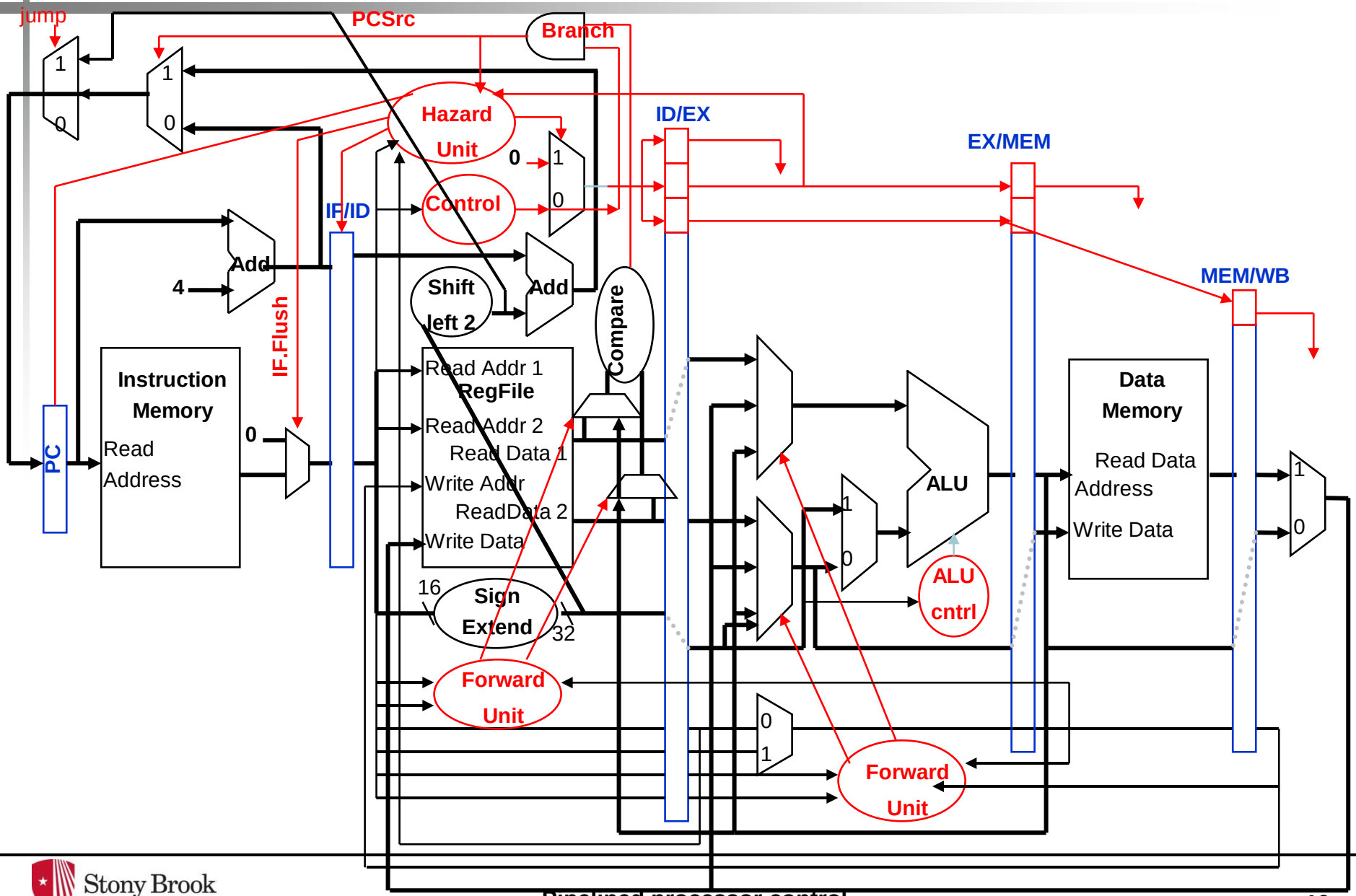
- Bypass of source operands from the EX/MEM

```
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
      ForwardC = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
      ForwardD = 1
```

Forwards the result from the second previous instr. to either input of the Compare

- MEM/WB "forwarding" is taken care of by the normal RegFile write before read operation
- If the instruction immediately before the branch produces one of the branch compare source operands, then a stall will be required since the EX stage ALU operation is occurring at the same time as the ID stage branch compare operation

Supporting ID Stage Branches

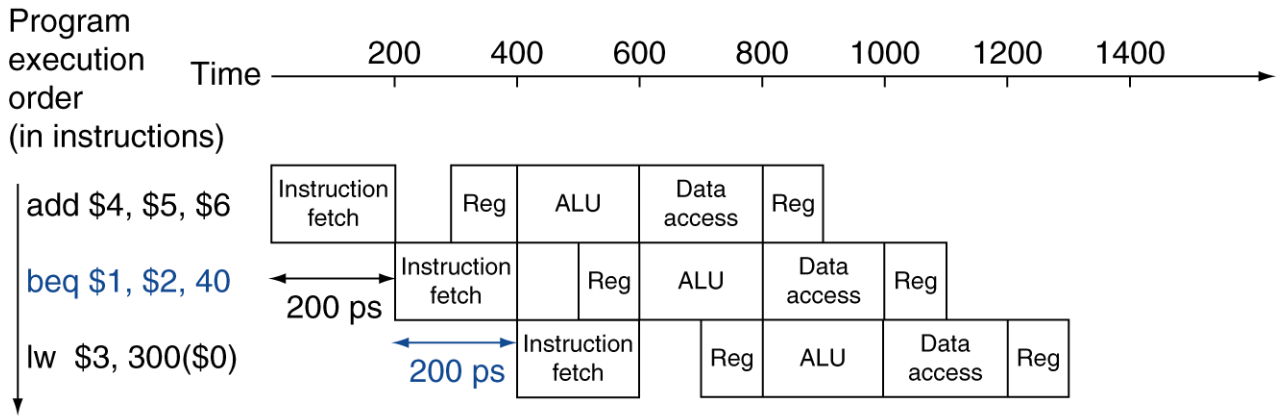


Branch Prediction

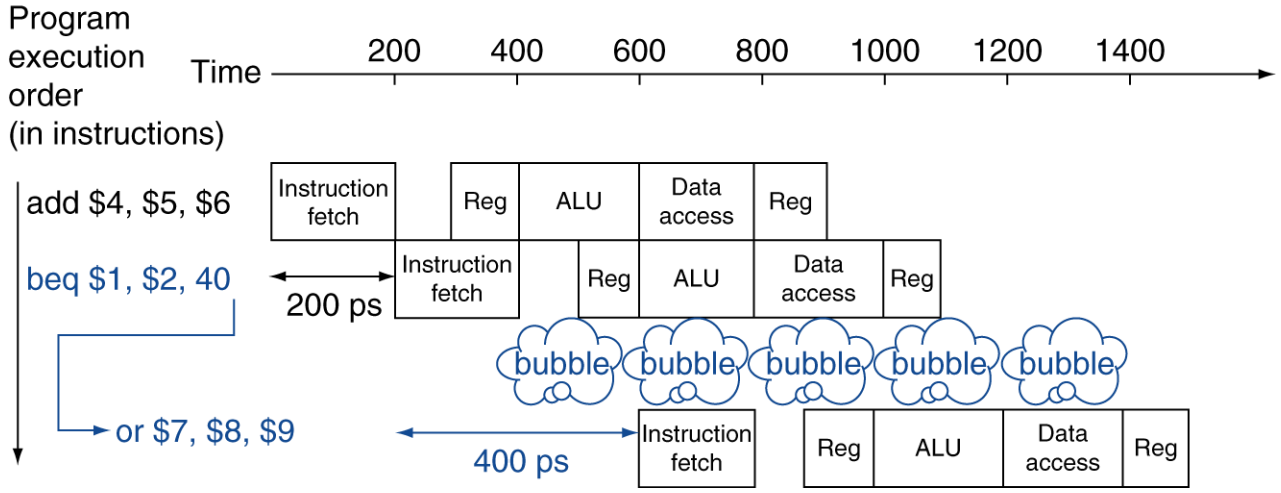
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect

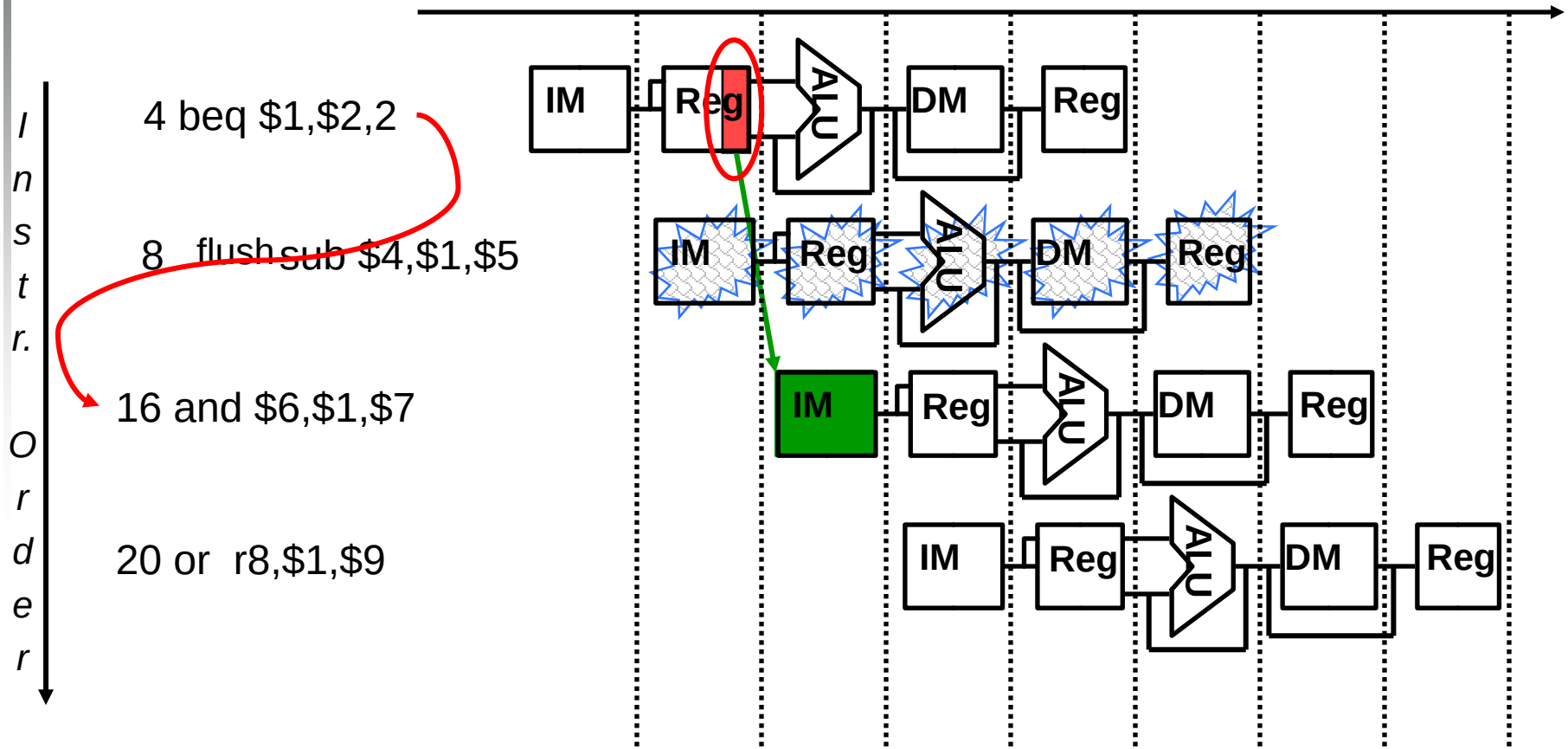


Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

1. Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch is taken does the pipeline stall
   - If taken, flush instructions in the pipeline after the branch
     - in IF, ID, and EX if branch logic in MEM – three stalls
     - in IF if branch logic in ID – one stall (one-cycle bubble)
   - ensure that those flushed instructions haven't changed machine state– automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite or RegWrite)
   - restart the pipeline at the branch destination

Flushing with Misprediction (Not Taken) when Branch is Taken



- To flush the IF stage instruction, add a IF.Flush control line that zeros the instruction field of the IF/ID pipeline register (transforming it into a nop)

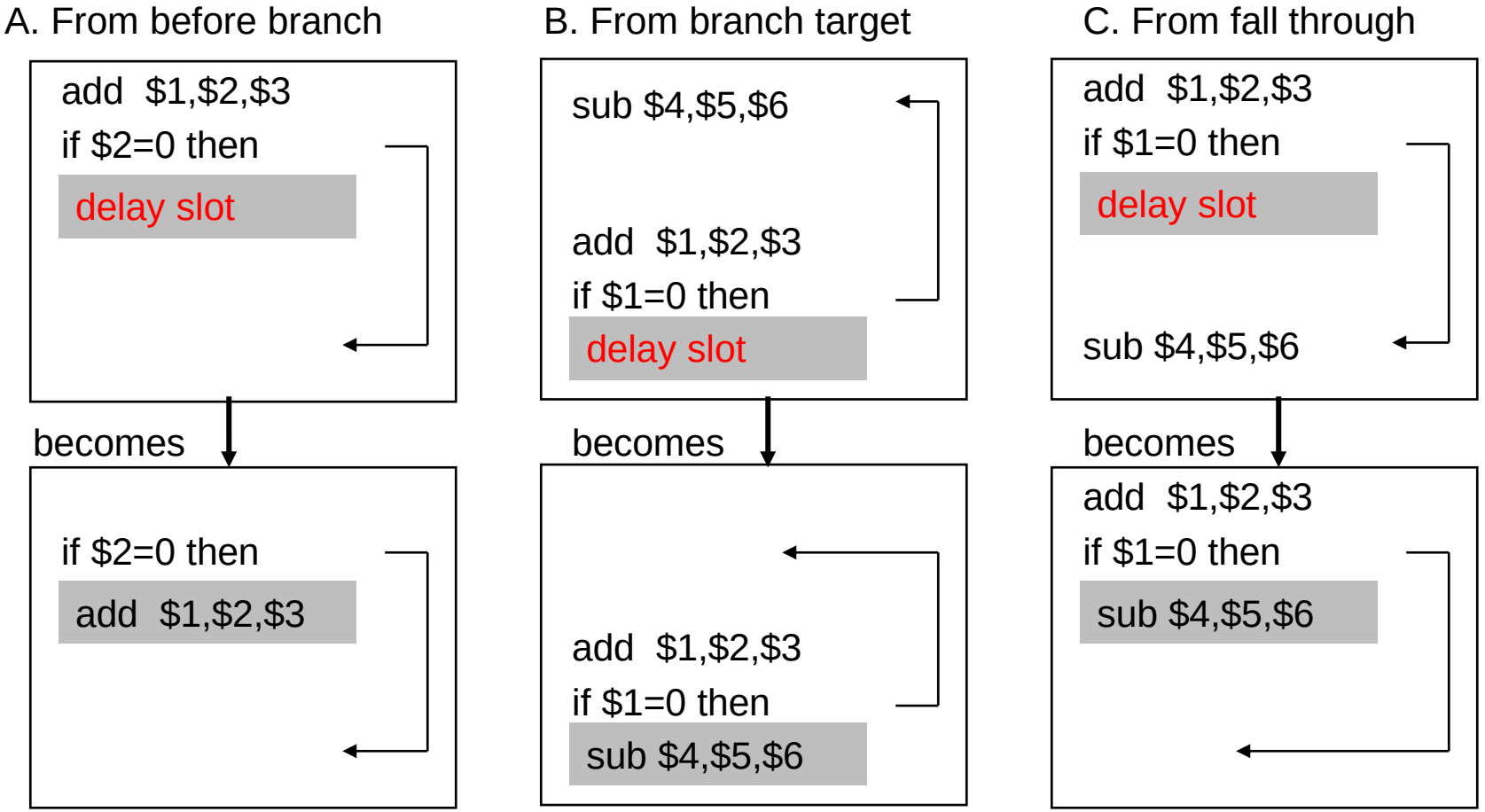
More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict forward branches not taken
    - Predict backward branches taken
      - **BUT** in MIPS Predict taken – always incurs one stall
        - if branch destination hardware has been moved to the ID stage
- Dynamic branch prediction (later in the lecture)
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Delayed Decision

- First, move the branch decision hardware and target address calculation to the ID pipeline stage
- A delayed branch **always executes the next sequential instruction** – the branch takes effect after that next instruction
  - MIPS software moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay

Scheduling Branch Delay Slots



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails

Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

If ideal CPI is 1, then:

$$\text{Pipeline Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

Pipelined vs. unpipelined

$$CPI_{pipelined} = Ideal\ CPI + \sum_{i=1}^{n} f_{hazard}(i) * penalty\ (stall\ cycles)\ (i)$$

Assume 4% unconditional branch, 6% conditional branch- untaken, 10% conditional branch-taken

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	3	1.52	3.3	1.0
Predict taken	1	1.20	4.2	1.27
Predict not taken	1	1.14	4.4	1.33
Delayed branch	0.5	1.10	4.5	1.38

When branch done in MEM stage → Stall pipeline

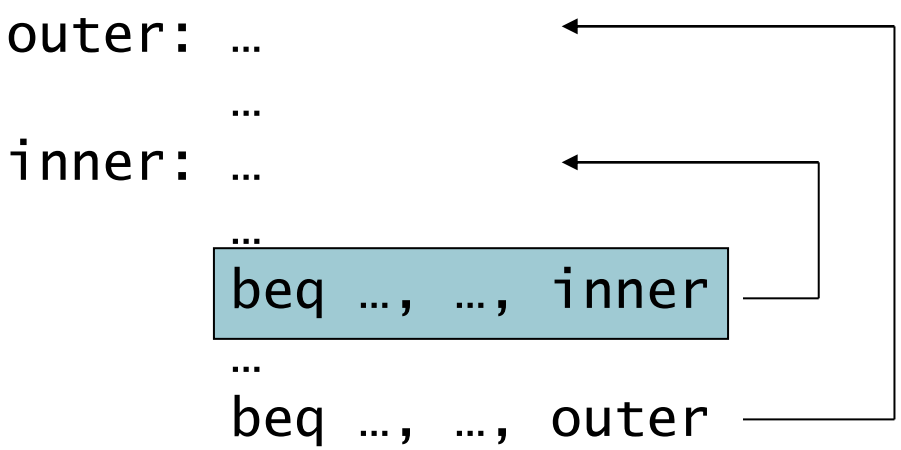
When branch done in ID stage → Predict taken, Predict not taken

Dynamic Branch Prediction

- In deeper and superscalar (multiple instructions/cycle) pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table (BHT) in IF stage
  - Indexed **by the lower bits of the PC branch instruction addresses**
  - Stores outcome (taken/not taken) in BHT
    - May predict incorrectly (may be from a different branch with the same low order PC bits, or may be a wrong prediction for this branch) but this doesn't affect correctness, just performance
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!



  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

1-bit Prediction Accuracy

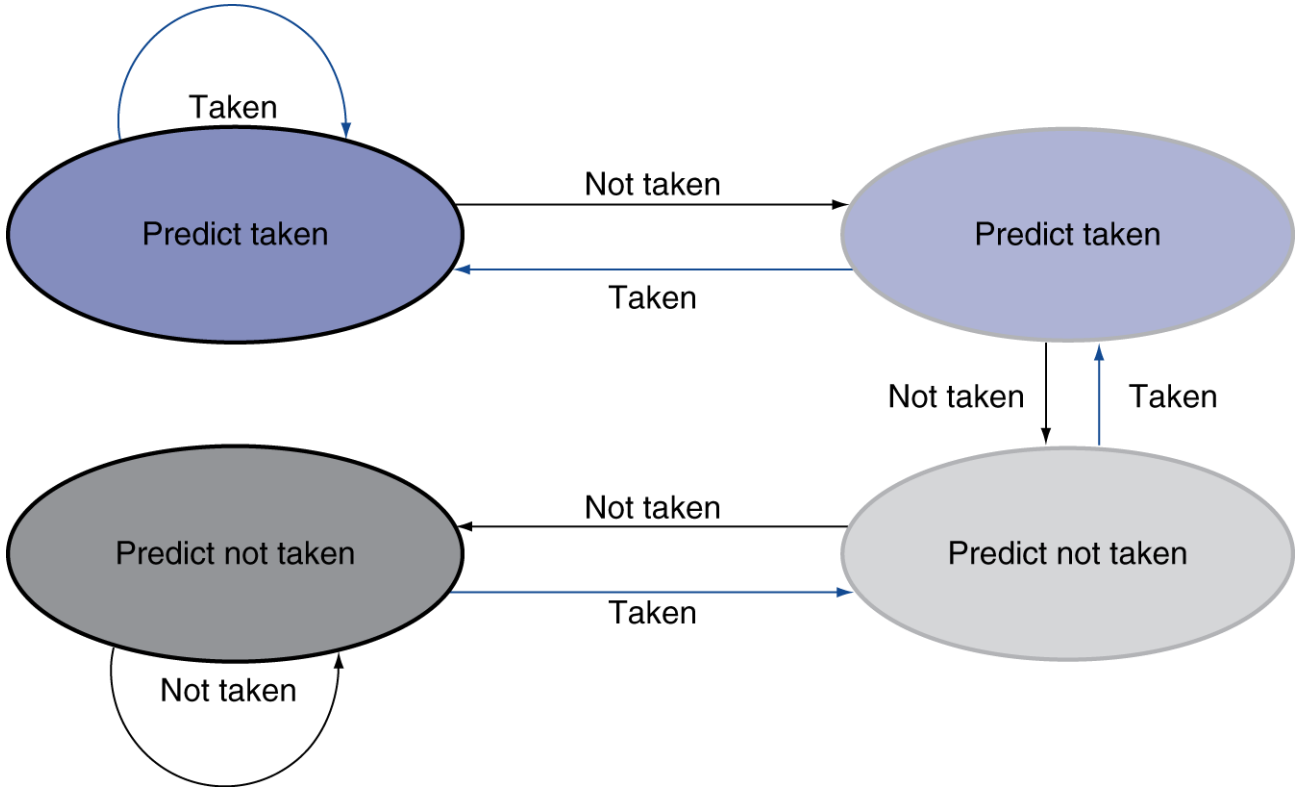
- 1-bit predictor in loop is incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
  2. As long as branch is taken (looping), prediction is correct
  3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

Loop: 1st loop instr
2nd loop instr
.
.
.
last loop instr
bne $1,$2,Loop
fall out instr

- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

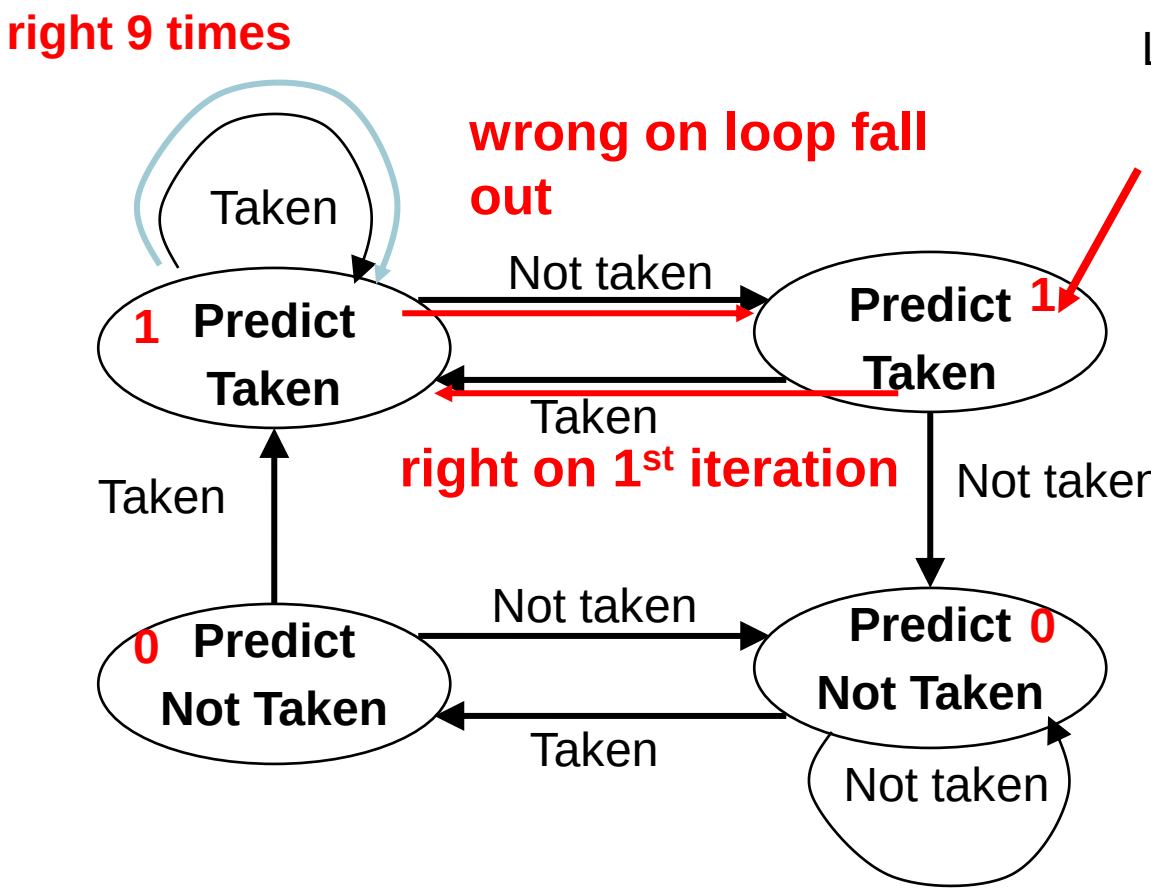
2-Bit Predictor

- Only change prediction on two successive mispredictions



2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed



Loop: 1st loop instr
2nd loop instr
.
.
.
last loop instr
bne $1,$2,Loop
fall out instr

Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer (BTB)
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

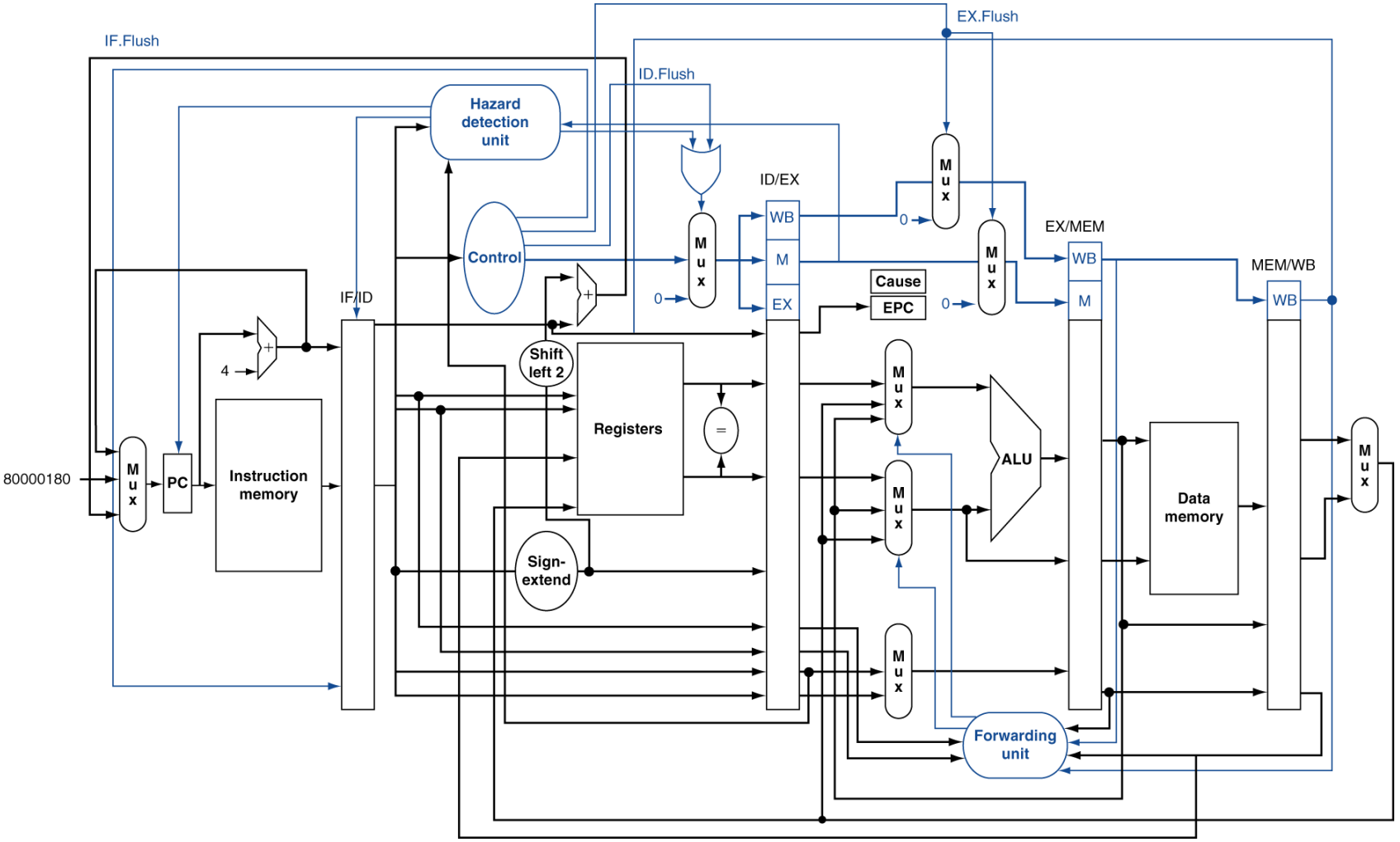
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

  `add $1, $2, $1`

  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

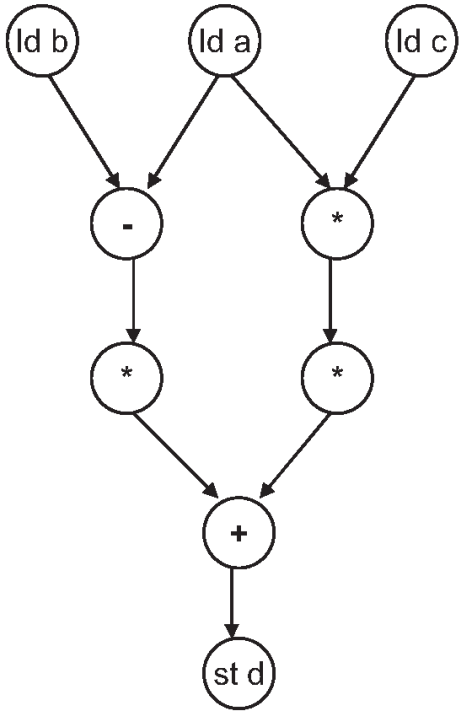
Instruction-level Parallelism (ILP)

$$D = 3(a - b) + 7ac$$

- **Sequential execution order**

  ld a
  ld b
  sub a-b
  mul 3(a-b)
  ld c
  mul ac
  mul 7ac
  add 3(a-b)+7ac
  st d

- **Data-flow execution order**



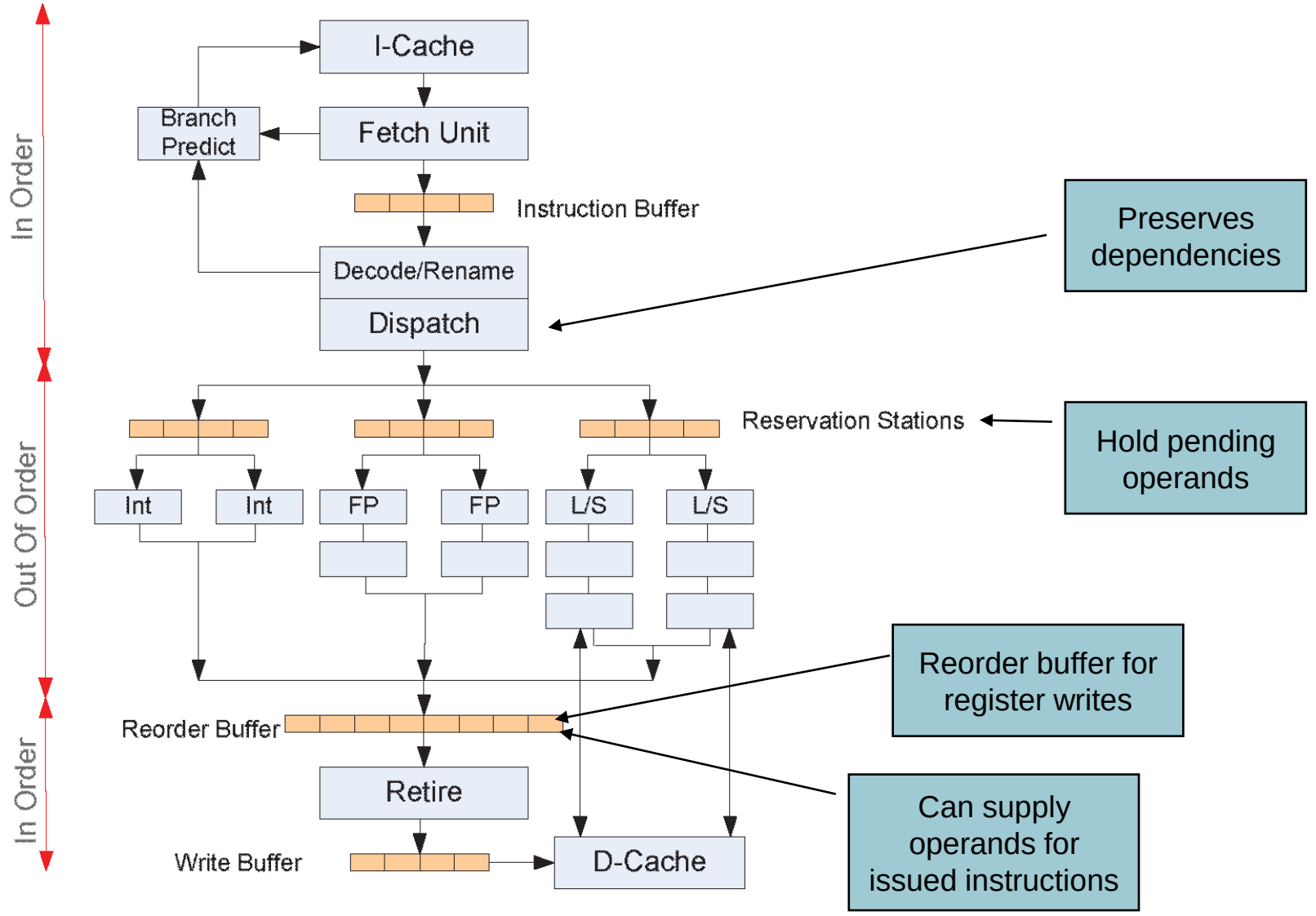
Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

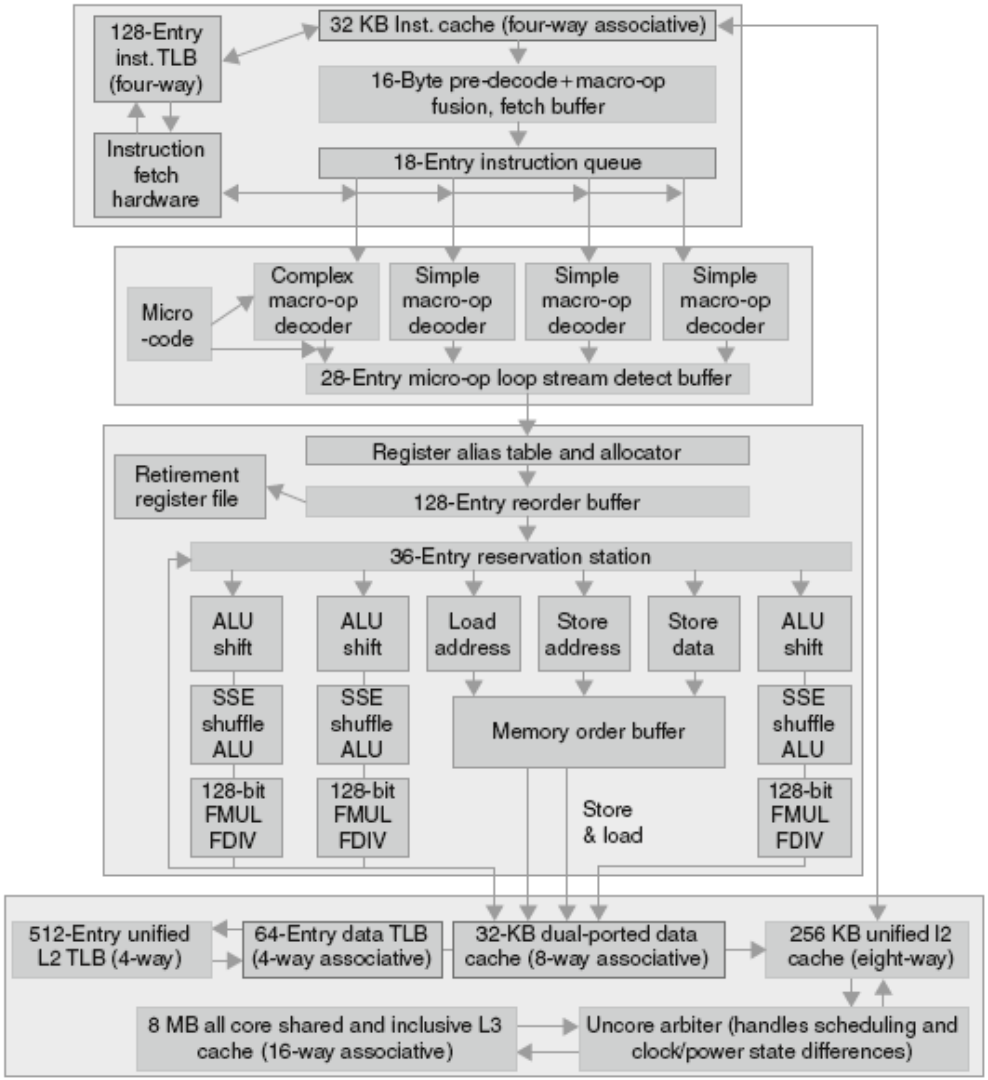
Multiple Instruction Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue in superscalar processors
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

A Modern Superscalar Out-of-Order Processor



Core i7 Pipeline



Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	3000 MHz	14	4	Yes	2	75 W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87 W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130 W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130 W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140 W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165 W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185 W

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)