
EECE 321: Computer Organization

Mohammad M. Mansour
Dept. of Electrical and Compute Engineering
American University of Beirut

Lecture 27: Pipelining
Control Hazards

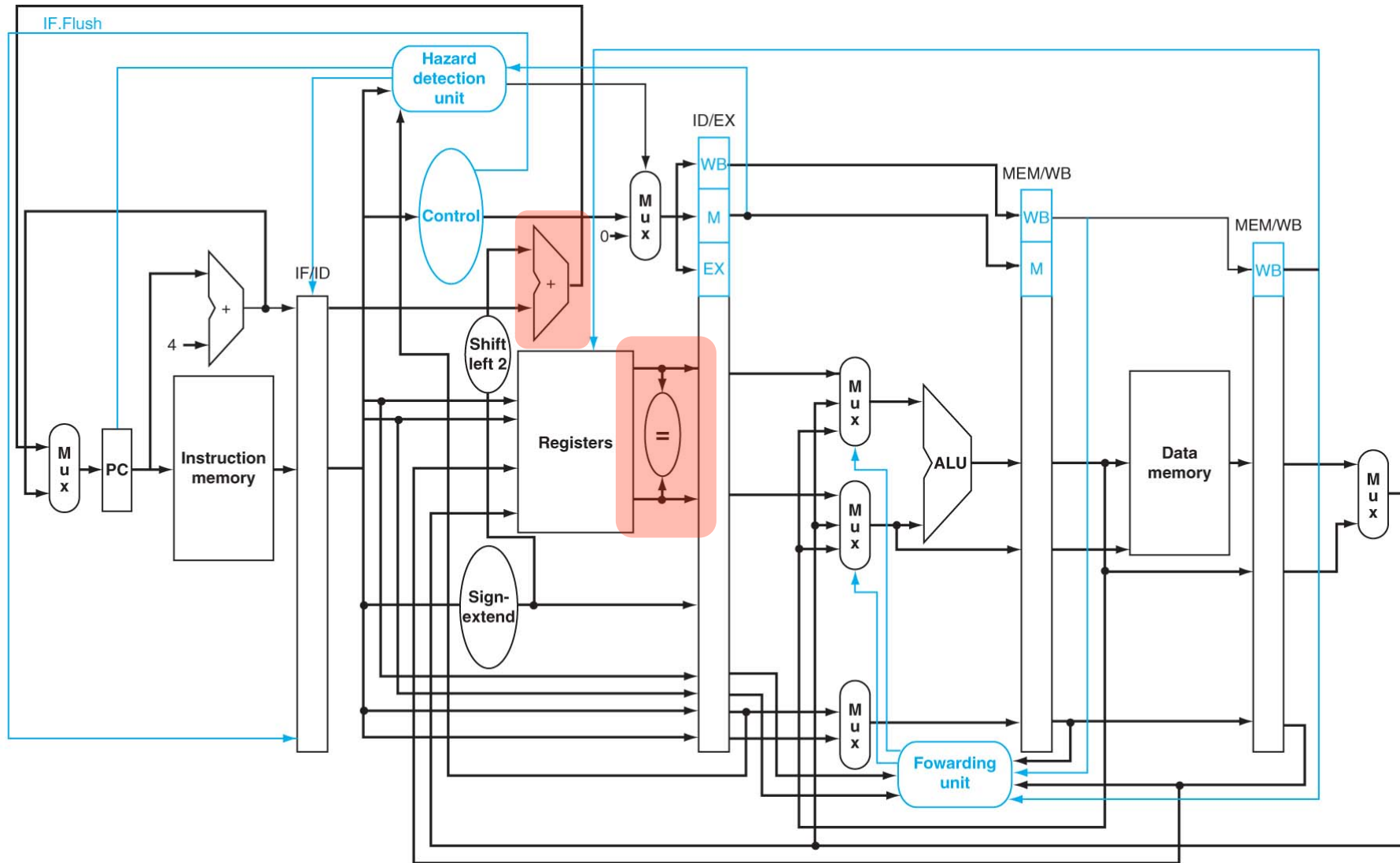
Announcements

- Exam II
 - Wednesday May 5 at 6:15pm
 - Rooms 543 & 545

- Final Exam
 - Monday May 31 from 9:00-12:00 noon
 - Wing D

Reducing the Delay of Branches

Assumption: Branches are resolved in ID. Hence need to flush IF stage in case we fetch from wrong target

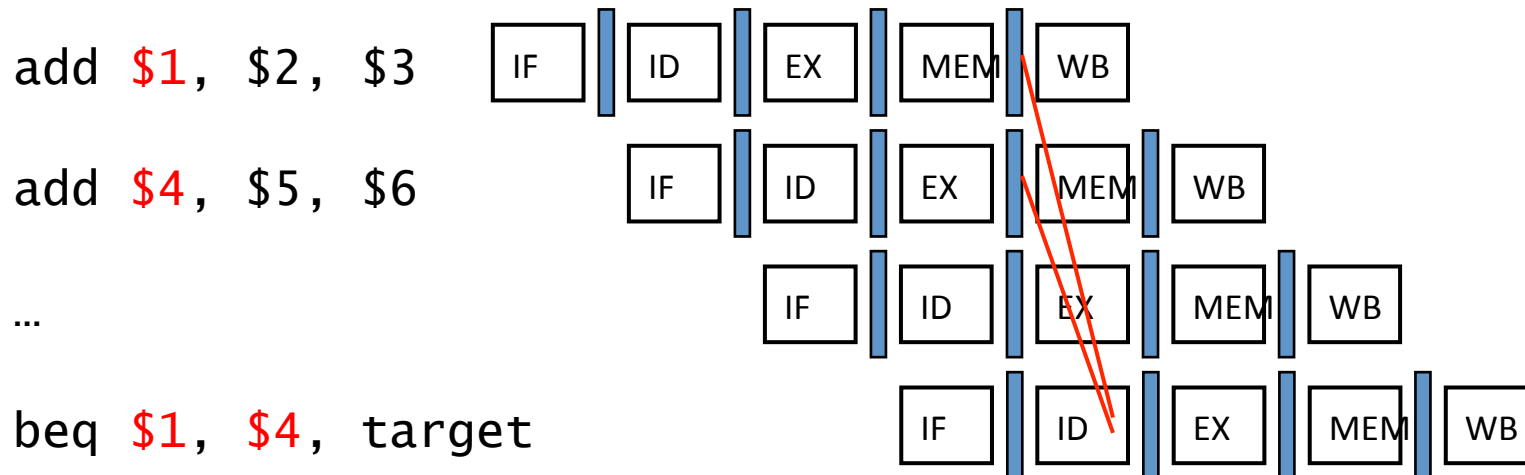


Data Hazards for Branches

- Notice now the branch instruction requires its operands in the ID stage!
 - So how to deal with data hazards related to branch operands?

- Example:

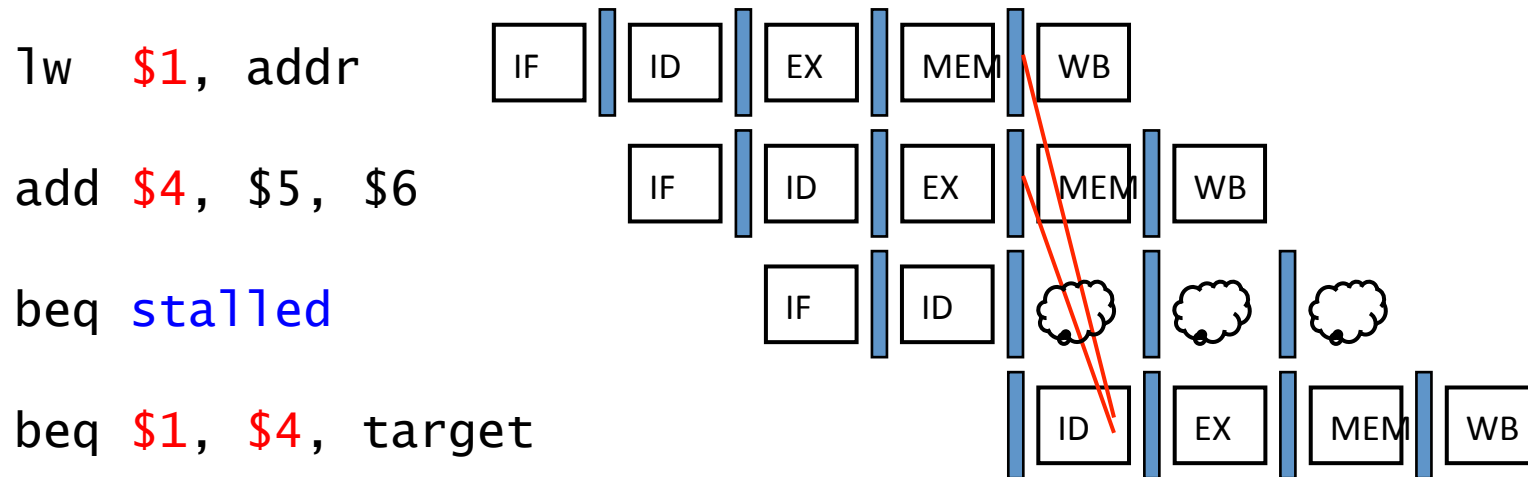
```
add $1, $2, $3
add $4, $5, $6
sub $6, $7, $8
beq $1, $4, target
```



- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction
 - Can resolve using forwarding

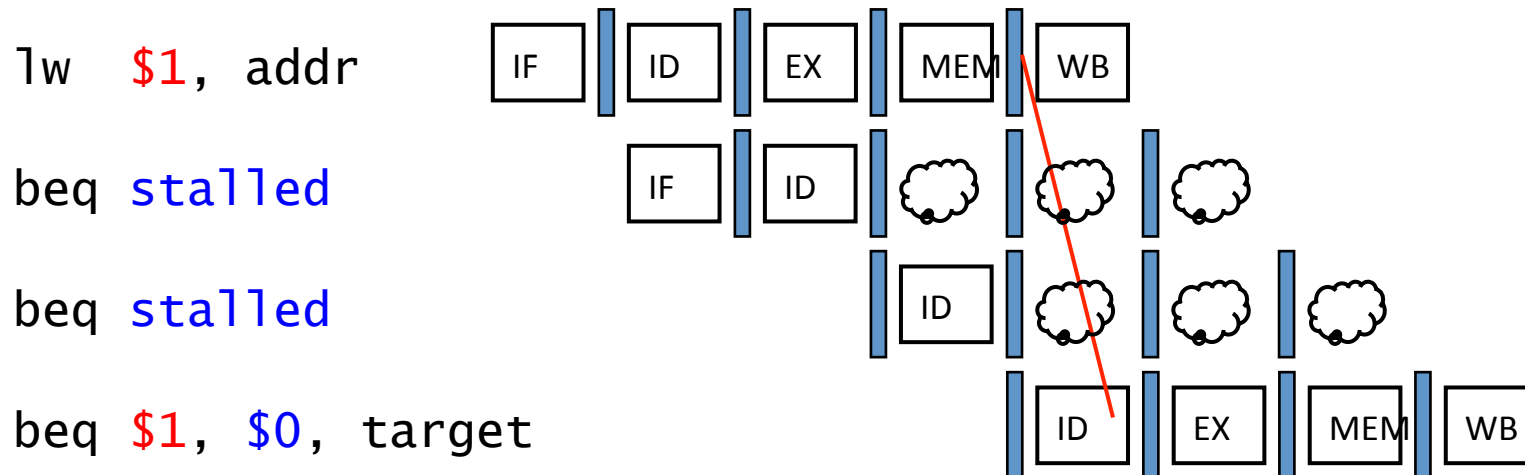
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
 - Need 1 stall cycle
- Example:



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
 - Need 2 stall cycles
- Example:

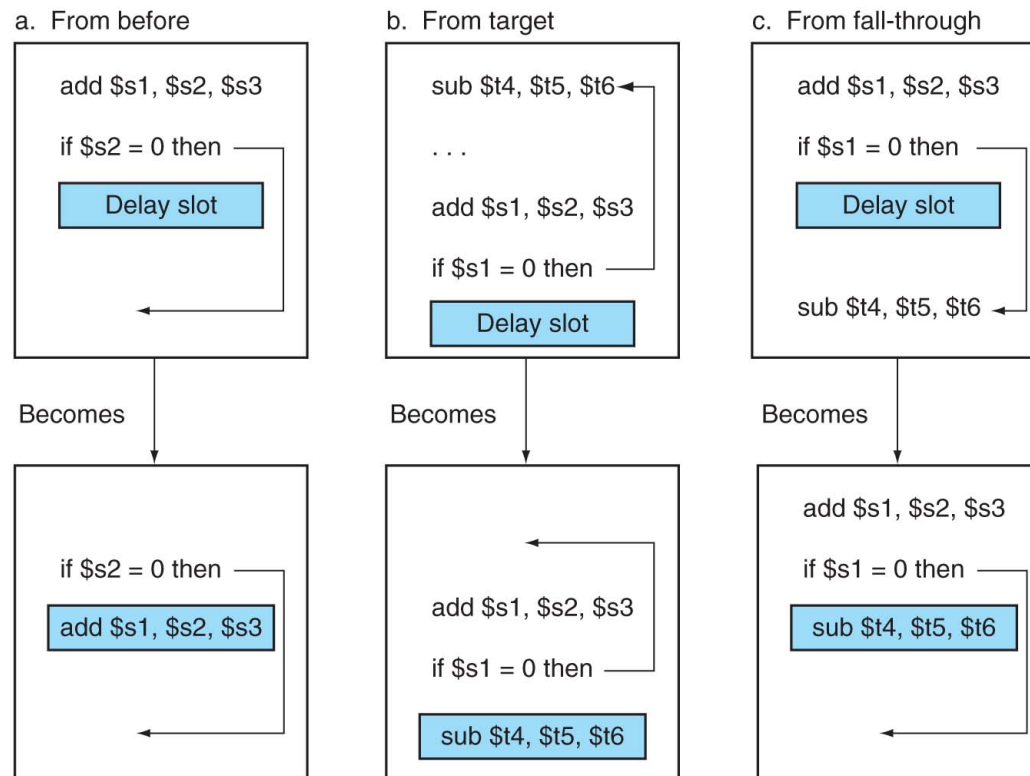


Branch Delay Slots

- So far we have considered only dynamic techniques (done by hardware) to resolve hazards
- What can the compiler do?
 - Called “static” techniques
- One technique is called Branch Delay Slot
- It is used by compiler (statically) to schedule instructions that always execute irrespective of the direction taken by the branch immediately after the branch
 - Compiler needs to know how many slots it has to fill
 - In our case, only one slot
 - In more realistic “deeper” pipelines, that is not the case. Compiler can’t always find such instructions to fill all slots.
 - Can be more creative: Schedule instructions that are “OK” to execute even if the branch goes the unintended way
 - Example: An instruction that updates a temporary register that is not used afterwards in case the branch goes the other direction!

Scheduling Techniques to Fill the Branch Delay Slots

- In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice.
- Strategies (b) and (c) are used when (a) is not possible.
 - Can't use the add instruction to fill the slot due to \$s1
 - To make this optimization legal for (b) or (c), it must be OK to execute the sub instruction when the branch goes in the unexpected direction.



Dynamic Branch Prediction (Done in Hardware)

- In deeper and superscalar pipelines, branch penalty is more significant
 - Can't just rely on compiler to fill the delay slots
- Need to use hardware techniques:
 - Dynamic prediction
- Assuming branch is not taken is one elementary form of branch prediction.
 - Example: Always “predict” branch is not taken.
- With more hardware, it is possible to improve the accuracy of our prediction
 - Monitor previous behavior of a branch and predict future behavior accordingly
 - Hardware needed:
 - Branch Prediction Buffer (BPB) or Branch History Table (BHT).
- **Branch Prediction Buffer:** A small memory indexed by the lower portion of the address of the branch instruction.
 - It contains a bit that says whether the branch was recently Taken or Not Taken.
 - Very simple; prediction not necessarily true; bit may have been set by another instruction that has the same low-order address bits.

Dynamic Branch Prediction (Done in Hardware)

- To execute a branch
 - Check branch prediction buffer, expect the same outcome
 - Start fetching from Fall-through or Target
 - If wrong, flush pipeline and flip prediction

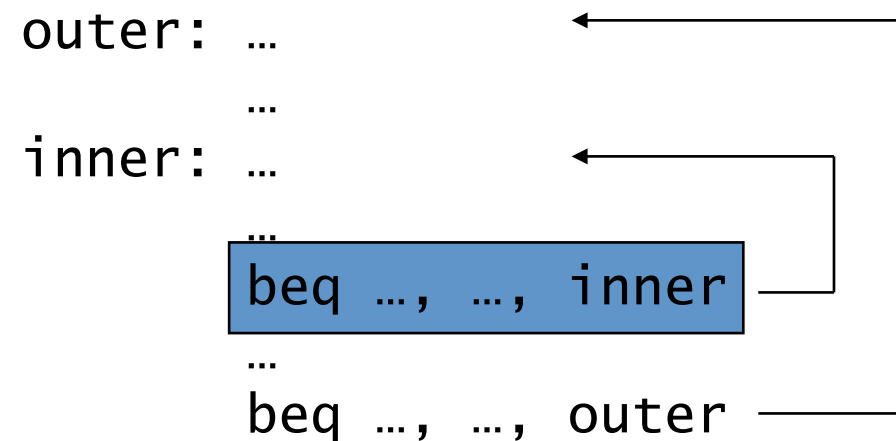
- Disadvantages of 1-bit prediction:
 - We will likely predict incorrectly twice rather than once when a branch is not taken.

- Example: Consider a loop that branches 9 times in a row, then it is not taken once. What is the prediction accuracy for this branch assuming a 1-bit prediction scheme.
 - Mispredict first and last loop iterations. Prediction accuracy is 80%.

- What about a 2-bit prediction scheme?
 - Prediction must be wrong twice before it is changed.

1-Bit Predictor: Shortcoming

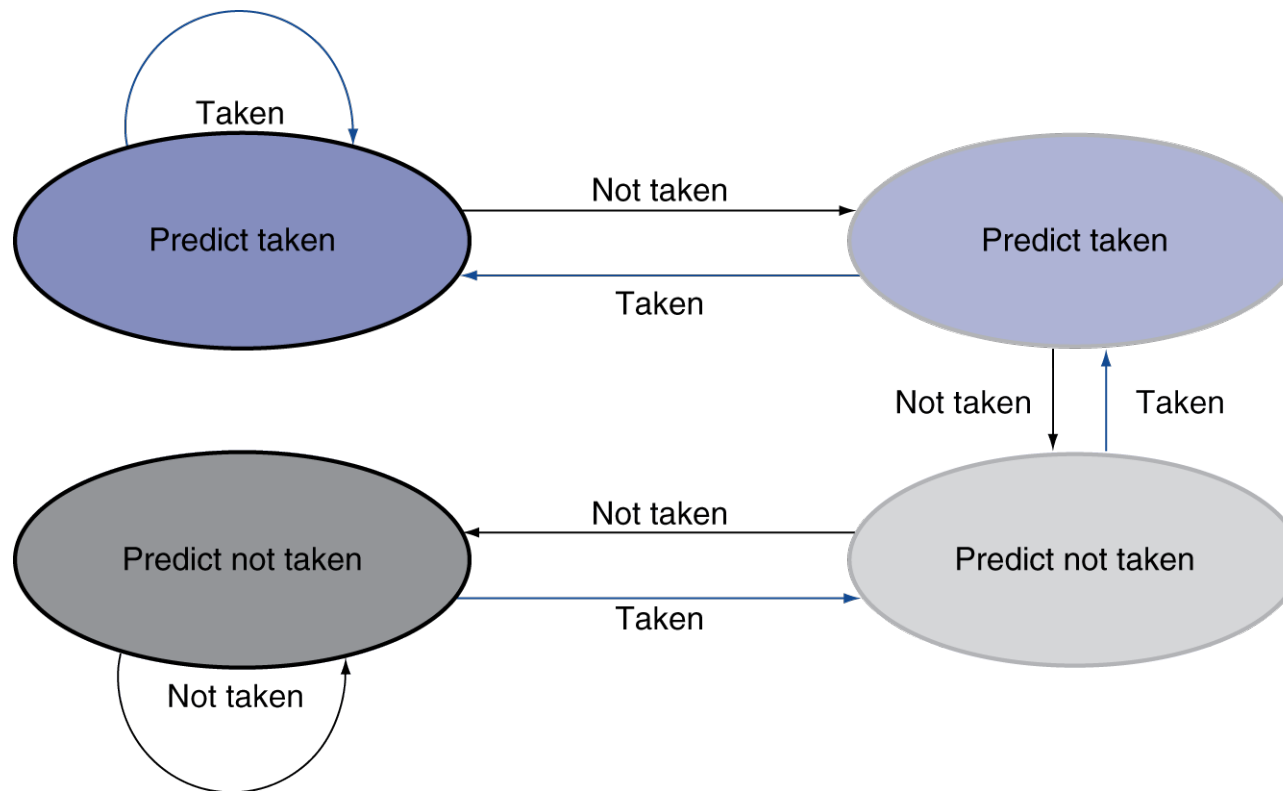
- Inner loop branches mispredicted twice!



- Mispredict as **Taken** on last iteration of inner loop
- Then mispredict as **Not Taken** on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions

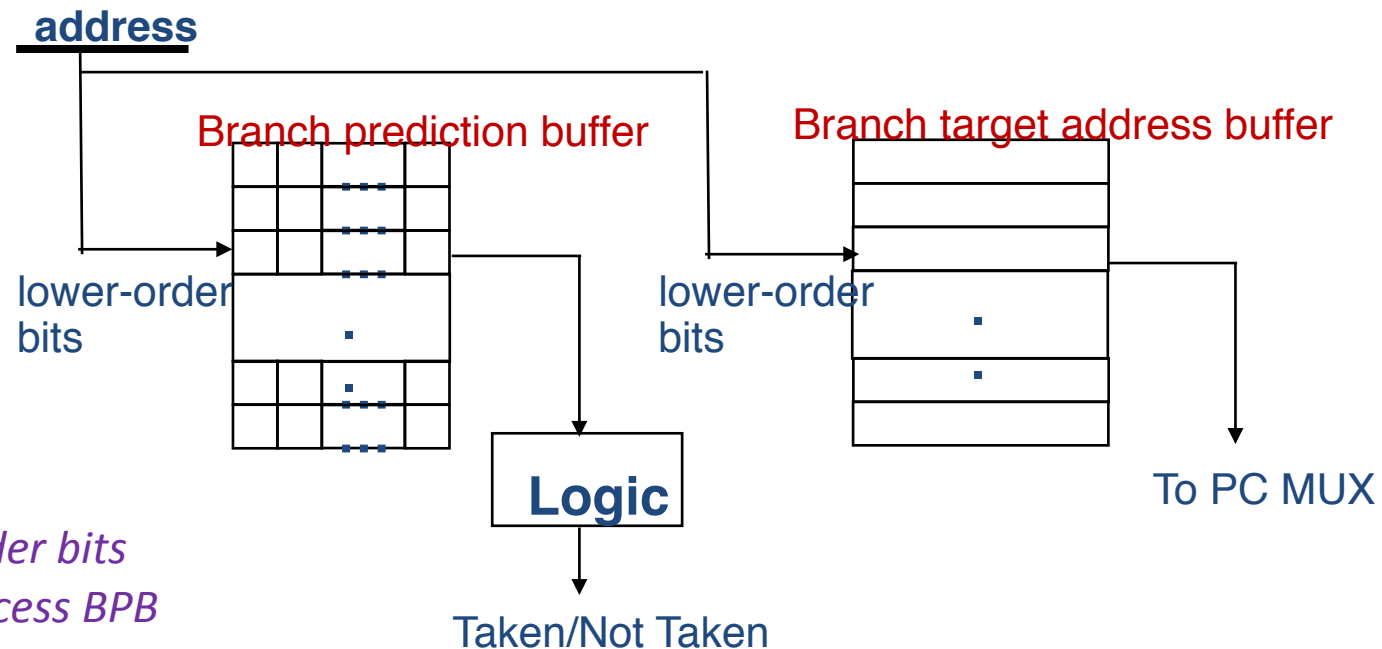


Calculating the Branch Target

- So far we have done the prediction part of the branch, but still need to target address in case the branch is taken
 - Where do we get this information from?
- If we don't have the target address ready in time, we need to stall
 - 1-cycle penalty for a taken branch
- Solution: Use a Branch Target Buffer (BTB)
 - It stores target addresses
 - Indexed by PC when instruction fetched
 - If hit and instruction is branch predicted taken, can fetch target immediately

Dynamic Branch Prediction

- The branch prediction buffer can be implemented as a small special buffer accessed with the lower-address bits of the instruction address during the IF stage.
- If the instruction is predicted as taken, the next instruction should be fetched from the target.
- How the BTB and BPB work together?



*Why lower-order bits
are used to access BPB
and BTB?*

Exceptions and Interrupts

Exceptions and Interrupts

- An exception is an unexpected event initiated from within the processor.
 - “Unexpected” event requiring change in flow of control
- Different ISAs use the terms *exceptions* and *interrupts* differently
- **Exception**
 - Arises within the CPU
 - Ex: undefined opcode/instruction, overflow, syscall, ...
- **Interrupt**
 - Initiated from an external I/O controller
 - Ex: Printer sends a “paper jam” interrupt, “out of paper” interrupt, etc
- Dealing with exceptions and interrupts without sacrificing performance is hard

Handling Exceptions

- In MIPS, exceptions are managed by a System Control Coprocessor (CP0)
- Steps taken to handle an exception:
 1. Save PC of offending (or interrupted) instruction
 - In MIPS: Exception Program Counter (EPC)
 2. Save indication of the problem
 - In MIPS: Cause register
 - For simplicity, we'll handle only two types of exceptions: *undefined opcode*, *overflow*
 - So use only a 1-bit cause register: 0 for *undefined opcode*, 1 for *overflow*
 3. Jump to a handler routine located at predefined address 0x8000 00180
 - Irrespective of exception type, always jump to same address to handle the exception
 - Then depending on the cause, the handler decides what to do further and where to jump

An Alternate Mechanism

- Vectored Interrupts
 - An alternative mechanism to handle exceptions
 - Handler address determined by the cause
 - So depending on cause, jump directly to the appropriate handler

- Example:
 - Undefined opcode: 0xC000 0000
 - Overflow: 0xC000 0020
 - ...: 0xC000 0040

- Handler instructions either
 - Deal with the interrupt, or
 - Jump to real handler

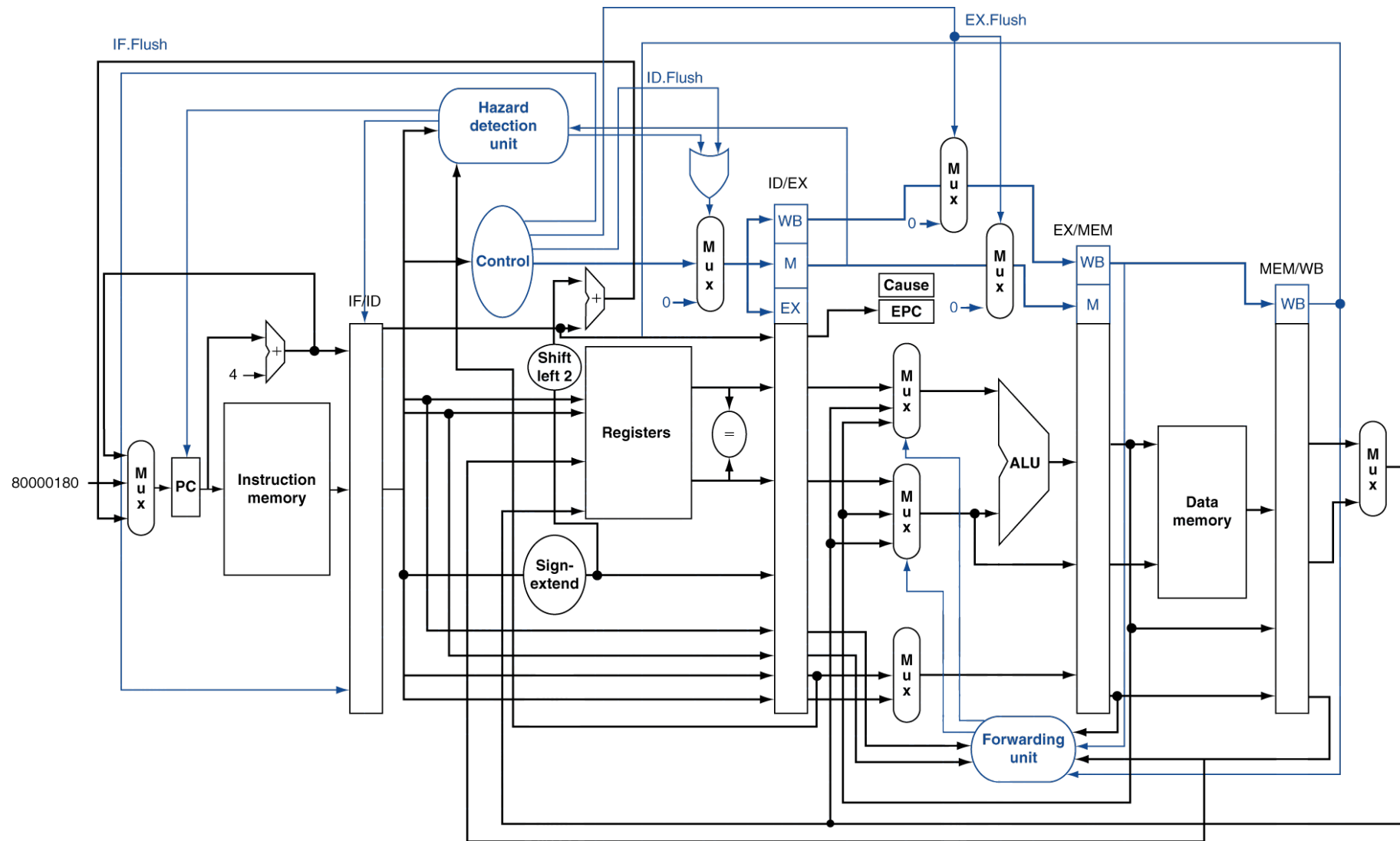
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If exception is “restartable”:
 - Take corrective action
 - Use EPC to return to program
- Otherwise
 - Terminate program
 - Report error using EPC, cause, ...

Exceptions in a Pipeline

- We treat them as another form of a “control hazard”
- Why?
- Consider overflow on add in EX stage
 `add $1,$2,$1`
 - Prevent \$1 from being clobbered. So the exception must be serviced directly after the instruction leaves the EX stage.
 - Must complete all previous instructions
 - So instructions in MEM and WB stages proceed normally
 - Flush `add` and subsequent instructions
 - Need to flush IF/ID, ID/EX, EX/MEM pipeline registers.
 - Set Cause and EPC register values
 - Save overflow bit from ALU in Cause register
 - Need to save PC (+4) in EPC
 - Transfer control to handler
 - Jump to 0x80000180
- Similar to mispredicted branch
 - Use much of the same hardware
- Upon servicing the exception and notifying the user with the offending instruction and the cause of the exception, the user/OS can elect to resume execution of the program.
 - Execute a “return from exception” (`rfe`) instruction which simply copies EPC to PC.

Pipelined Datapath with Exception Handling



Exception Properties

- Restartable exceptions
 - Pipeline can flush the instruction
 - Handler executes, then returns to the instruction
 - Re-fetched and executed from scratch

- PC saved in EPC register
 - Identifies causing instruction
 - Actually PC + 4 is saved
 - Handler must adjust

Example

- Consider the following instruction sequence:

40hex **sub** \$11, \$2, \$4

44hex **and** \$12, \$2, \$5

48hex **sub** \$13, \$2, \$6

4Chex **add** \$1, \$2, \$1

← **causes overflow exception**

50hex **slt** \$15, \$6, \$7

54hex **lw** \$16, 48(\$7)

...

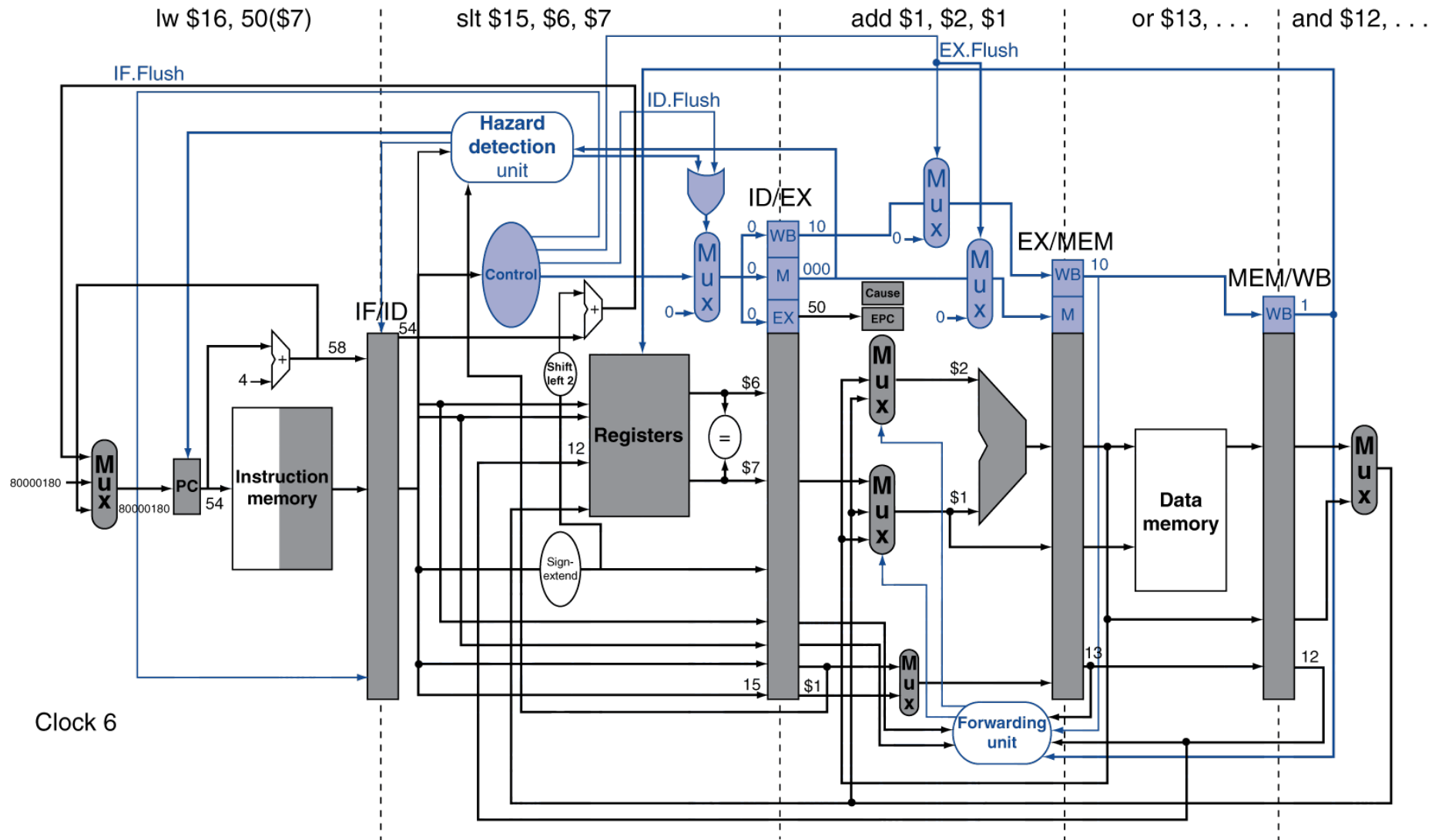
- Assume the instructions of the exception service routine to be invoked on an exception begin like this:

80000180hex **sw** \$25, 1000(\$0)

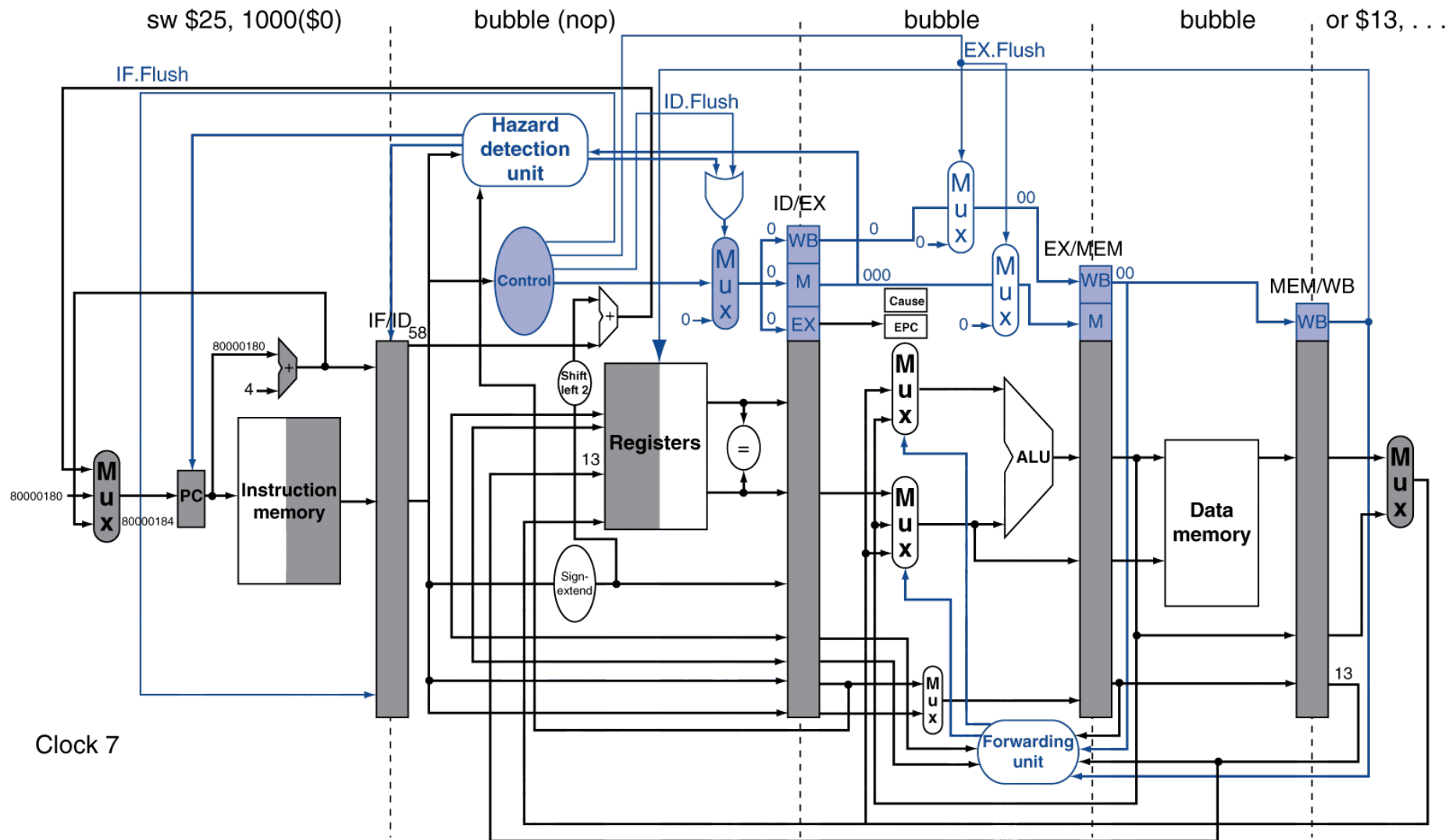
48000184hex **sw** \$26, 1004(\$0)

...

Example



Example



Multiple Exceptions

- Multiple exceptions can occur simultaneously in the pipeline.
- What if the following happens:
 - An add instruction overflows in EX stage
 - An undefined instruction is identified in ID stage
- Simple approach: deal with exception from earliest instruction
 - Flush subsequent instructions
 - Upon returning from exception, the undefined instruction after add is serviced.
 - This approach is called “Precise” exceptions
- In complex pipelines however,
 - Multiple instructions issued per cycle
 - Out-of-order completion occurs
 - So maintaining precise exceptions is difficult!