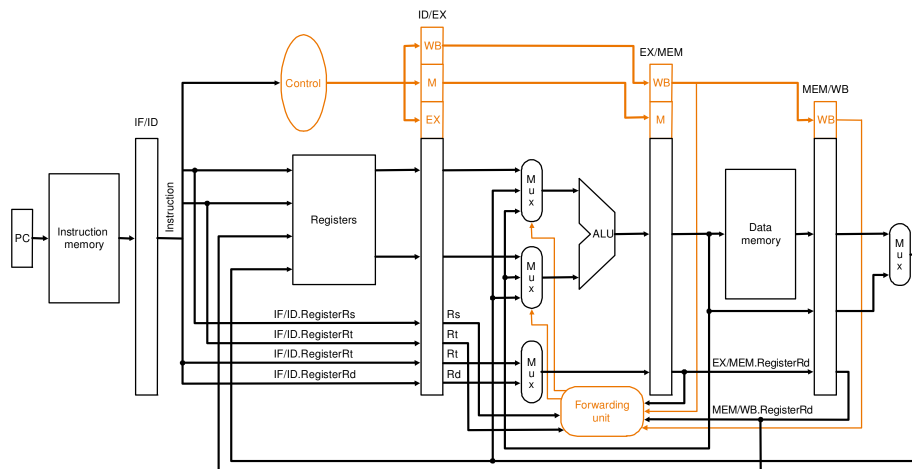
Pipelining: Implementation (3)

Lecturer: Dr. Hui Annie Guo
huig@cse.unsw.edu.au
K17-501F (ext. 57136)

Overview

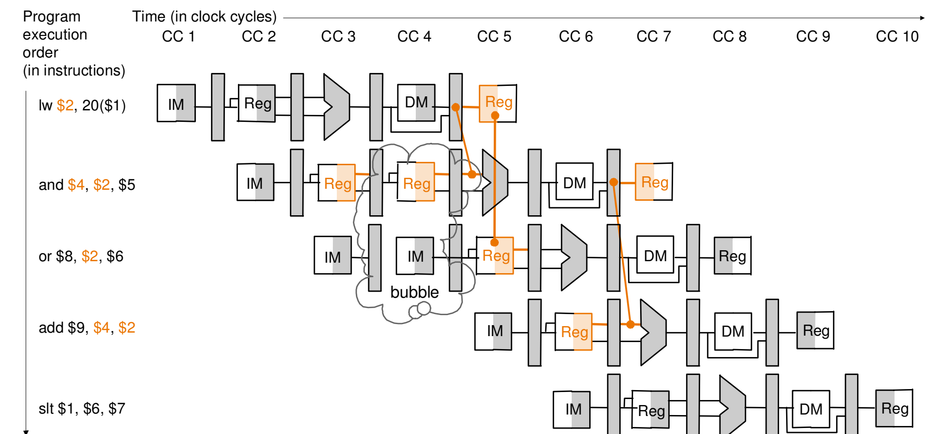
- Data hazards and design solutions (cont.)
- Control hazards and design solutions

Recall: Datapath with forwarding unit



When forwarding does not work ⇒ STALL

- Detect hazard
- Stall at ID stage



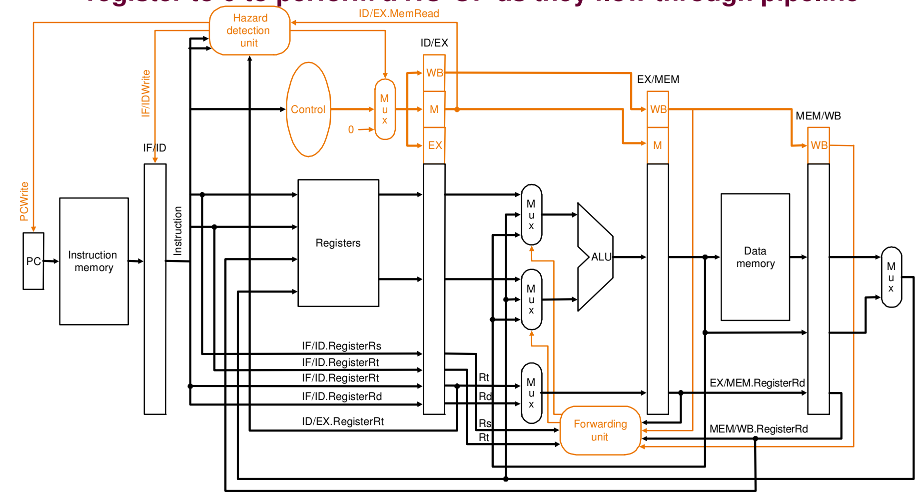
Detecting hazards

- When a load instruction is followed by an instruction dependent on the memory data (load-use hazard)
- We add a hazard detection unit that checks for the following condition:

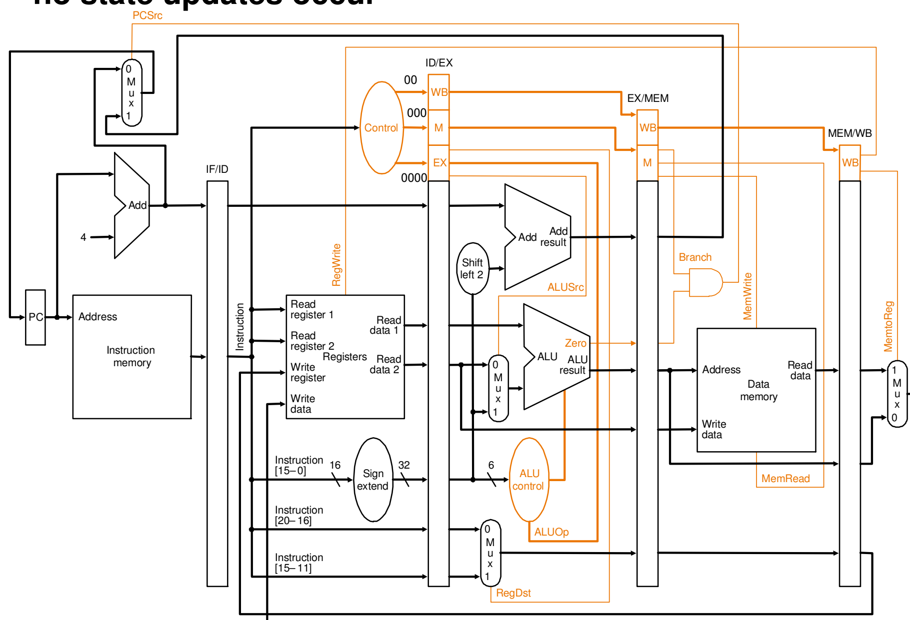
IF ID/EX.MEMRead
AND (ID/EX.RegisterRt = IF/ID.RegisterRs
OR ID/EX.RegisterRt = IF/ID.RegisterRt)
stall the pipeline

How to stall the pipeline?

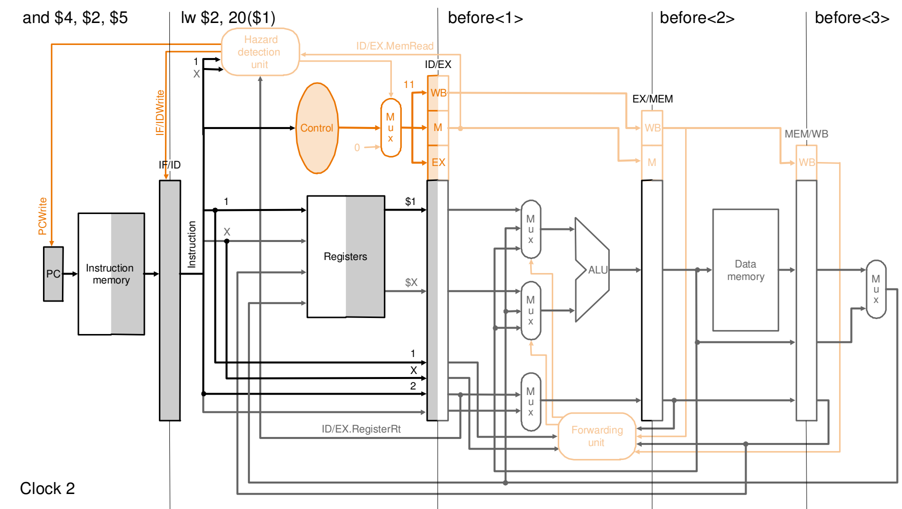
- Prevent the PC and IF/ID pipeline register from changing by not loading new values
- Set the EX, MEM and WB control fields of the ID/EX pipeline register to 0 to perform a NO-OP as they flow through pipeline



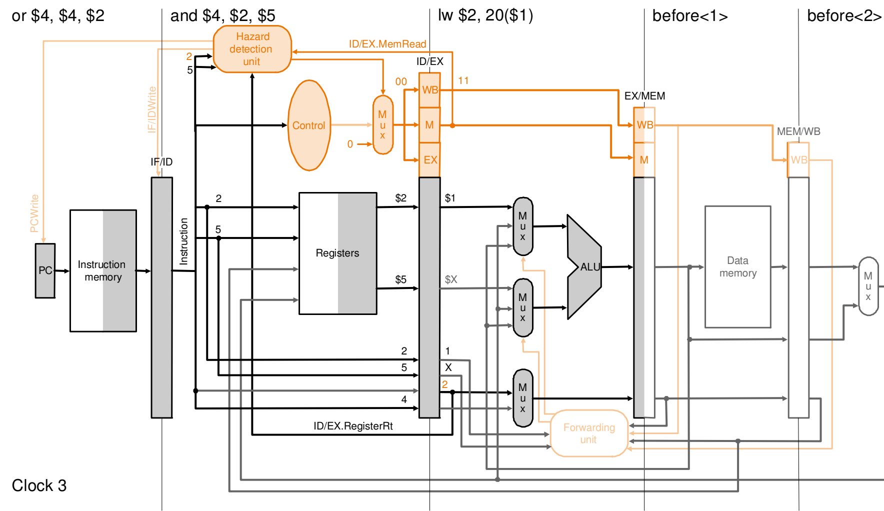
Setting the control signals to 0 does nothing (nop) since no state updates occur



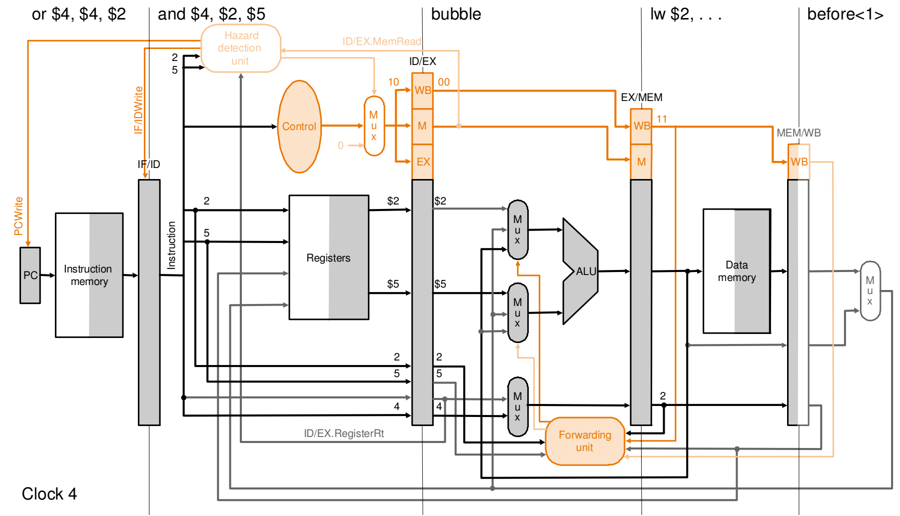
Load-use hazard (LUH) example



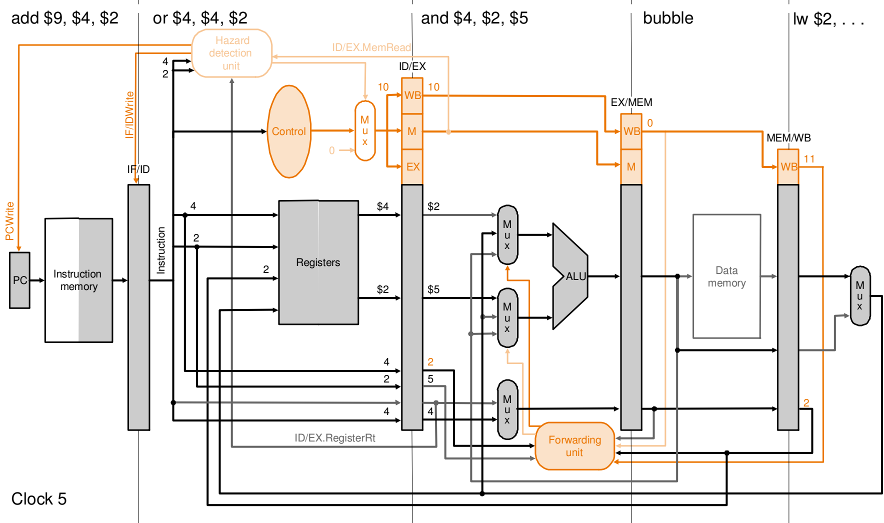
LUH example (cont.)



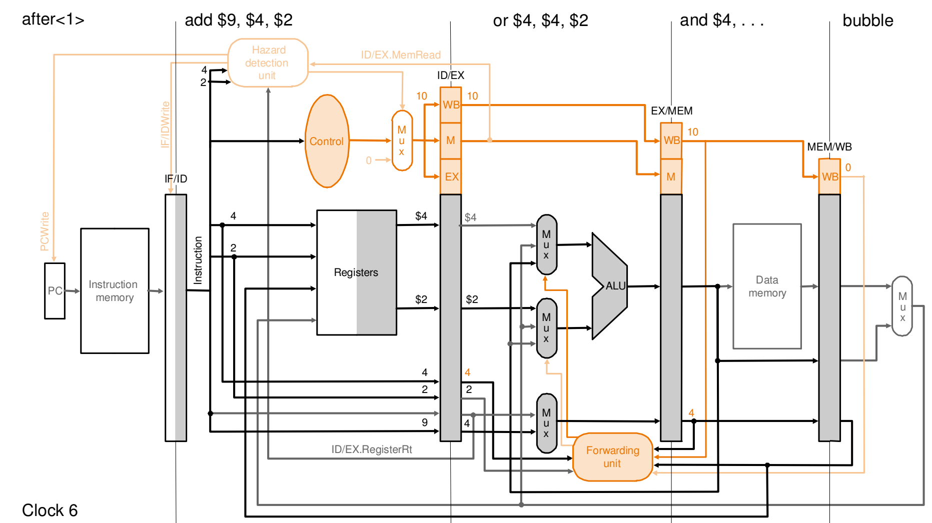
LUH example (cont.)



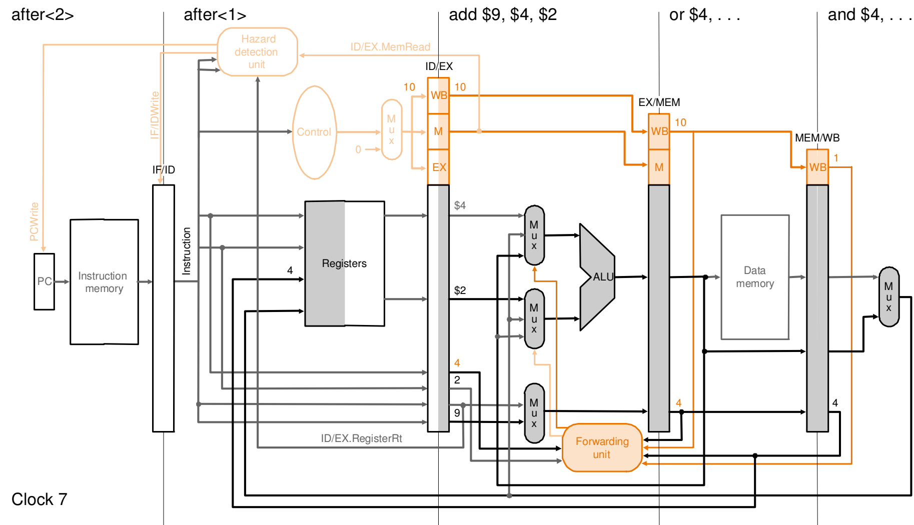
LUH example (cont.)



LUH example (cont.)



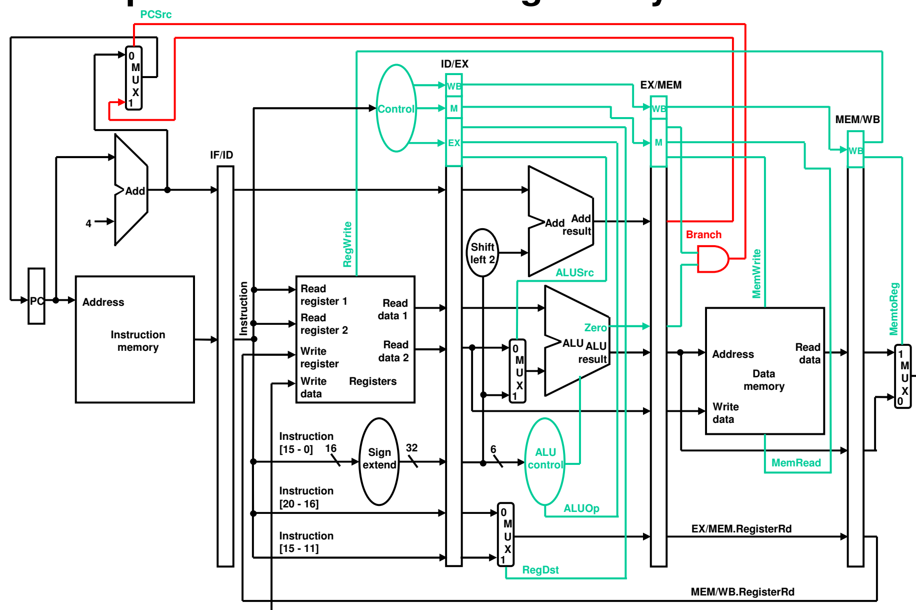
LUH example (cont.)



Control hazards

- Delay in determining the proper instruction to fetch is a control hazard or branch hazard
- Solutions to control hazards
  - Stall
    - Wait until the branch decision is clear
      - Up to three cycles will be wasted for each branch depending upon the design of the datapath (availability of resources)
  - Branch prediction
    - static
    - dynamic
  - Branch delay

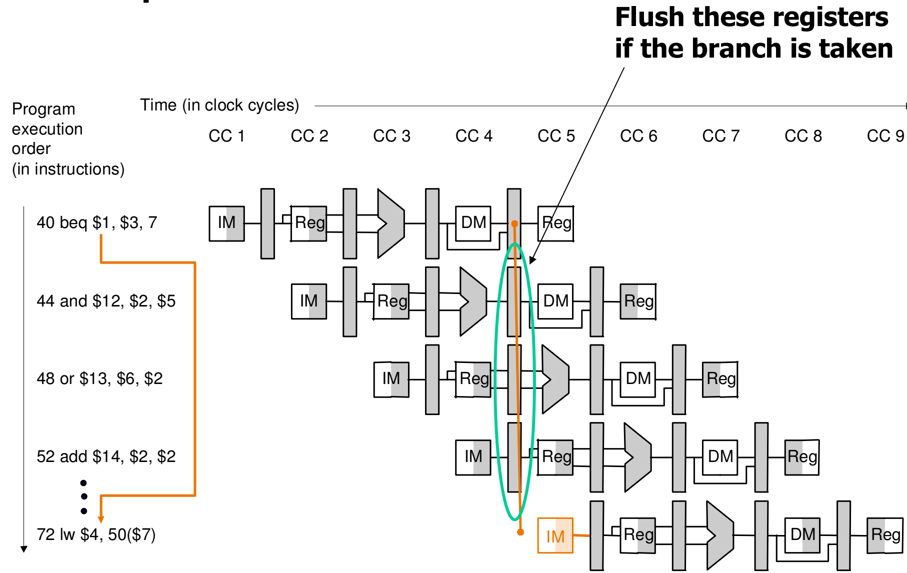
PC updated with branch target in cycle 4



Static prediction

- Predict that the branch displays a same behavior
  - Never taken
  - Always taken
- When the prediction is wrong, the instructions being fetched, decoded, and executed after the branch instruction, must be discarded (flushed)
- When the prediction is correct, the pipeline continues processing
- Better than stall
  - Assume the result of the branch is computed during the EX stage and the PC is updated during the MEM stage
  - If the accuracy of the prediction is x, then there is 3/(3-2x) speedup for branch instructions

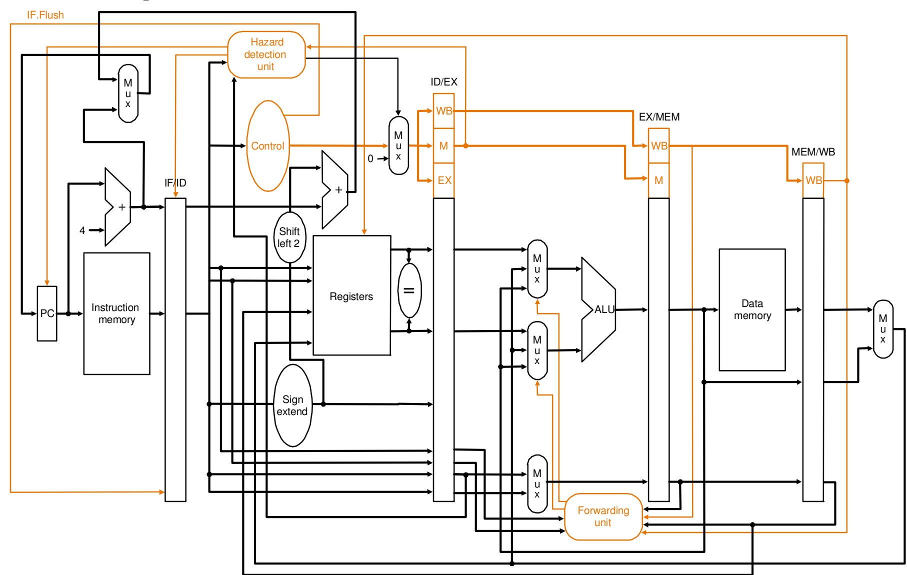
Static prediction – Branch not taken



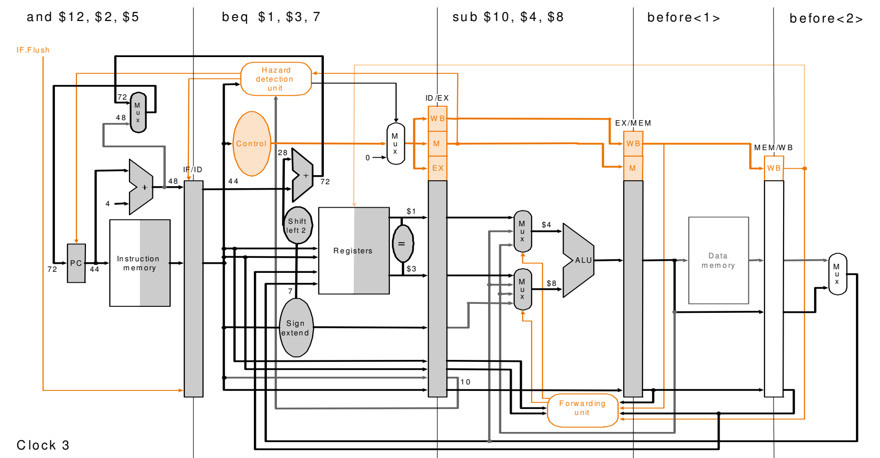
Static prediction – Branch not taken (cont.)

- **Reducing the delay of branches – flushing 3 instructions is a heavy penalty!**
  - **Make branch decision earlier**
    - **Calculate the target address in the ID stage**
    - **Compare register contents in the ID stage**
      - **use bitwise XOR (FAST, since no carry logic required)**
  - **Require forwarding to the ID stage and hazard detection, if the branch is dependent upon the result of an R-type or LOAD instruction that is still in the pipeline**
  - **Only one instruction then needs to be flushed**
  - **The control signal *IF.Flush* is used to flush the instruction in the IF/ID register**
    - **Sets the instruction field of IF/ID register to 0 (nop)**

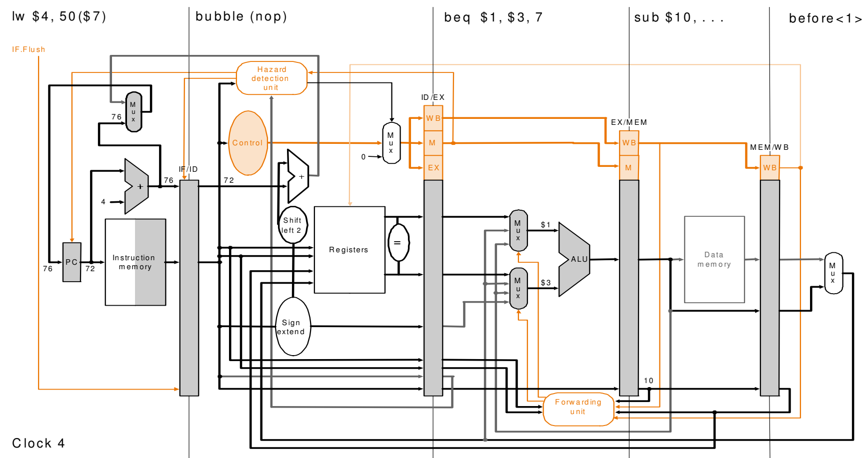
Static prediction – Branch not taken (cont.)



Handling misprediction(1)



Handling misprediction(2)



Dynamic prediction

- The prediction is made on the fly, depending on the history of the branch behavior
- The history is stored in a table or buffer, called the *branch history table* or *branch prediction buffer*
- The table consists of a portion of the branch instruction address and a branch history bit field

branch prediction buffer

Branch history

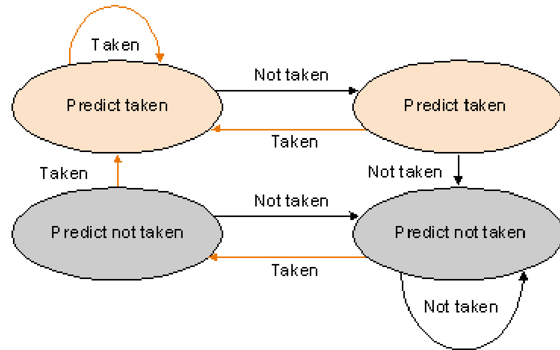
branch instruction address

1-bit prediction scheme

- The history bit field contains only 1 bit
- Prediction takes the bit value recorded from the last evaluation
  - If the prediction is wrong, the bit is changed
- Advantage
  - Simple
- Disadvantage
  - Double mis-prediction
    - Consider a loop that is executed several times
      - With 1 bit scheme, we always predict incorrectly when we exit the loop
      - But, when we execute loop again, we mispredict on the first iteration since the history bit was set to “not taken” at the end of the last execution
  - A 2-bit scheme is often used to overcome this problem

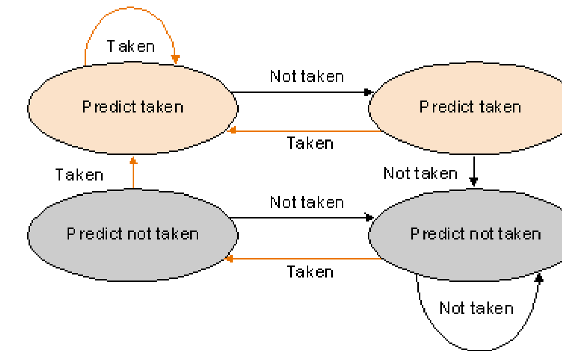
2-bit prediction

- History field has two bits
- A prediction must be wrong twice before it is changed
- Only mispredict once per loop execution



How to implement it?*

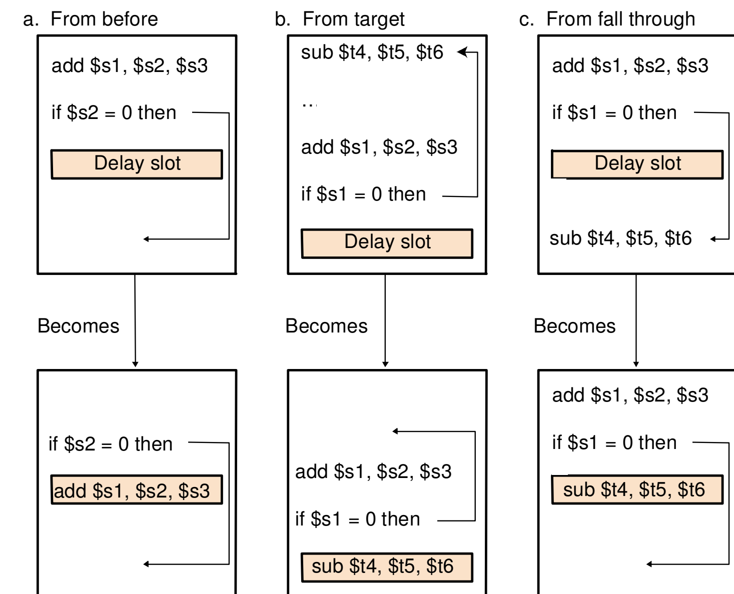
- Encoding the four states
- The state values are stored in the two bits of the history table
- The MSB indicate the prediction, 0 for TAKEN, 1 for NOT TAKEN



Delaying a branch

- A compiler approach
- Avoids the uncertainty of a branch decision by inserting an independent instruction (immediately after the branch instruction) that can always be executed irrespective of the branch decision (does not need to be flushed)
- Is becoming less popular as pipelines become longer and issue more instructions per cycle
- There is a preference to implement dynamic predictors instead

Scheduling a branch delay slot*



Pipelining review

- What is pipelining?
  - allow multiple instructions to be executed in a datapath simultaneously
- Pipelined datapath vs single cycle datapath
  - Similarities
    - CPI = 1 (ideal pipelining case)
    - Control signals are generated once per instruction
  - Differences
    - Clock cycle time of the pipelined datapath is smaller
    - Control signals of the pipelined datapath need to be buffered
- Pipelined datapath vs multicycle datapath
  - Similarities
    - Same cycle cycle time if same datapath partitions
    - Multiple cycles per instruction
  - Differences
    - Multiple instructions concurrently executing in the pipelined datapath
    - CPI (multi-cycle datapath) > CPI (pipelined datapath)

Pipelining review (cont.)

- Problems with pipelining
  - Resource competition
    - Structural hazards
  - Dependencies between instructions
    - Data hazards
    - Control hazards
- Solutions
  - For structural hazards
    - Resource replication
      - E.g. IMEM and DMEM
    - Structural design to avoid potential resource competition
      - E.g. Bypass in MEM stage

Pipelining review (cont.)

- Solutions
  - For data hazards
    - Hardware approaches
      - Forwarding
        - » From EX/MEM (ALU output)
        - » From MEM/WB (MEM output)
      - Stall + forwarding
        - » Load-read: stall read instruction one clock cycle then forwarding data from MEM/WB
    - Software approaches/enhancements
      - Make the two dependent instructions X number of instructions apart
        - » X >=3, no data hazards
        - » X = 2, no data hazards with proper register write and read control
        - » X = 1, no data hazards with forwarding
        - » X = 0, no data hazards for R-type instructions with forwarding; one cycle stall for load-read instructions

Pipelining review (cont.)

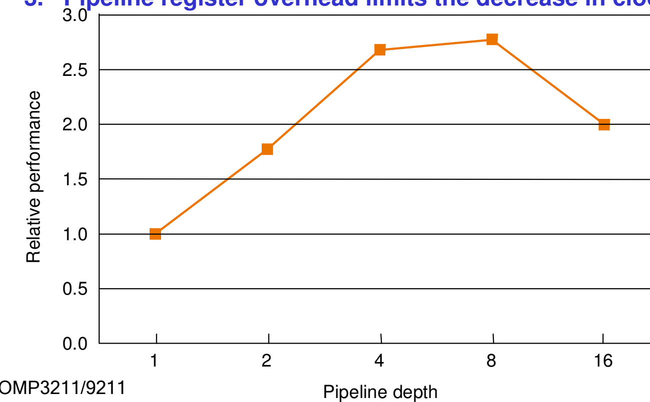
- Solutions
  - For control hazards
    - Hardware approaches
      - Evaluate and calculate target address earlier
        - » Done in ID stage
      - Prediction
        - » Branch taken/not taken (static and dynamic)
        - » Flush if wrong
    - Software approaches
      - Delay branch

Beyond pipelining

- **To achieve greater performance, pipelining concepts have been extended in 3 ways:**
  1. **Super-pipelining**
     - **longer pipelines, e.g. 8 or more stages**
  2. **Multi-issue pipelining**
     - **replicate datapath components so that multiple instructions can be launched each cycle**
     - **Allow instruction execution rate to exceed the clock rate, or for CPI < 1**
  3. **Dynamic pipelining (dynamic pipeline scheduling)**
     - **Avoids pipeline hazards by bringing forward instructions that otherwise wait on a dependency to be resolved**
     - **Requires a more sophisticated pipeline control and instruction execution model**

Increasing pipeline depth does not always increase performance

- **Three factors limit performance improvement gained by increasing pipelining depth**
  1. **Data hazards in the code increase the time per instruction because a larger percentage of cycles become stalls**
  2. **Control hazards result in slower branches**
  3. **Pipeline register overhead limits the decrease in clock period**

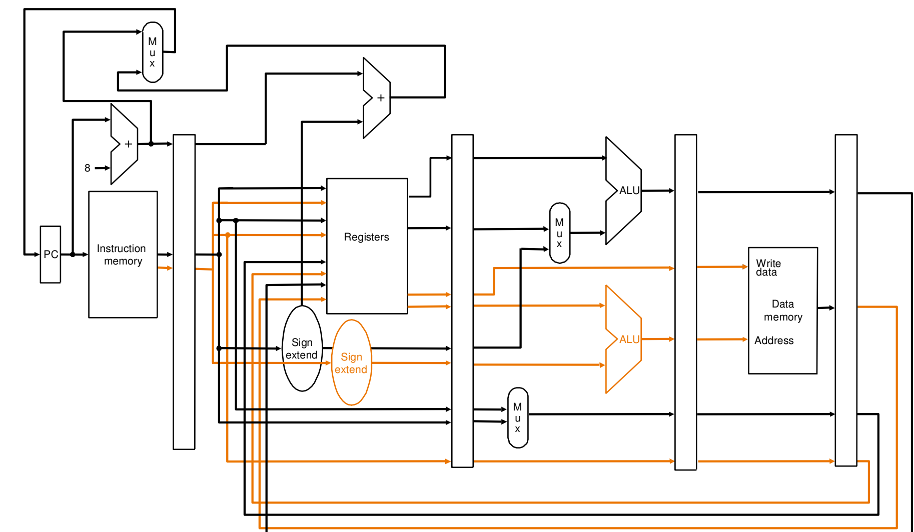


What does a multi-issue pipeline implementation look like?

- **Assume 2 instructions issued per clock cycle**
  - **One an integer ALU op or branch, the other a load or store**
  - **Hence, fetch 64 bits of instruction per cycle & assume they are aligned such that the ALU op always appears first**

Instruction Type	Pipe Stages						
ALU/branch	IF	ID	EX	MEM	WB		
Load/store	IF	ID	EX	MEM	WB		
ALU/branch		IF	ID	EX	MEM	WB	
Load/store		IF	ID	EX	MEM	WB	
ALU/branch			IF	ID	EX	MEM	WB
Load/store			IF	ID	EX	MEM	WB

A static 2-issue datapath



Multi-issue pipeline hazards

- In order to avoid structural hazards, we
  - Need extra read & write ports on the register file to allow ALU and data transfer ops to be issued in parallel
  - Need an additional adder to calculate memory addresses for data transfers
- Data hazards arising from loads cause a stall of two instructions
  - Need to employ more ambitious compiler & hardware scheduling techniques

Simple multi-issue code scheduling

- The following loop needs to be reordered to make use of the multi-issue resources:

```
Loop: lw    $t0,0($s1)    # $t0=array element
      addu  $t0,$t0,$s2   # add scalar in $s2
      sw    $t0,0($s1)    # store result
      addi  $s1,$s1,-4    # decrement pointer
      bne   $s1,$0,Loop   # branch $s1!=0
```

- The reordered sequence is:

```
CC          ALU/branch inst        Data transfer
1   Loop:                          lw    $t0,0($s1)
2           addi  $s1,$s1,-4
3           addu  $t0,$t0,$s2
4           bne   $s1,$0,Loop      sw    $t0,4($s1)
```

- Note just one pair of instructions executes in multi-issue mode – it takes 4 cycles for 5 instructions – CPI = 0.8 > 0.5!

Dynamic pipelining*

- Dynamic pipeline scheduling goes past stalls to find later instructions to execute while waiting for the stall to be resolved
- Typically, the pipeline is divided into an instruction fetch and issue unit, several (5-10) execute units, and a commit unit