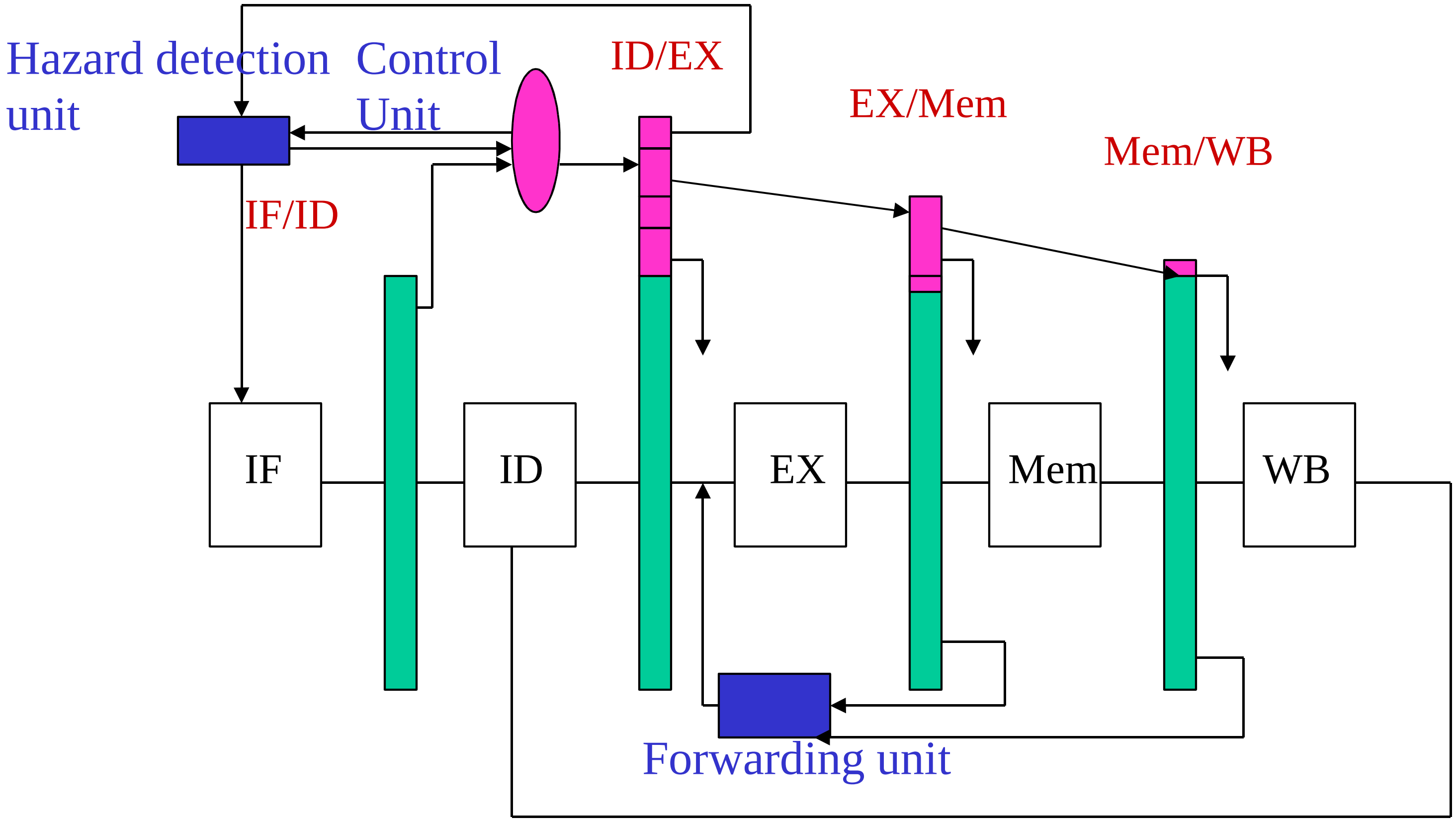
Control unit extension for data hazards



Forwarding unit

- Forwarding is done prior to ALU computation in EX stage
- If we have an R-R instruction, the forwarding unit will need to check
  - whether EX/Mem result register = IF/ID rs
  - EX/Mem result register = IF/ID rt
  - and if so set up muxes to ALU source appropriately
- and also whether
  - Mem/WB result register = IF/ID rs
  - Mem/WB result register = IF/ID rt
  - and if so set up muxes to ALU source appropriately

Forwarding unit (ct'd)

- For a Load/Store or Immediate instruction
  - Need to check forwarding for rs only
- For a branch instruction
  - Need to check forwarding for the registers involved in the comparison

Forwarding in consecutive instructions

- What happens if we have

  add $10,$10,$12

  add $10,$10,$12

  add $10,$10,$12

  Forwarding priority is given to the most recent result, that is the one generated by the ALU in the EX/Mem, not the one passed to Mem/Wb

  – So same conditions as before for forwarding from EX/MEM but when forwarding from MEM/WB check if the forwarding is also done for the same register from EX/MEM

Hazard detection unit

- If a Load (instruction *i-1*) is followed by instruction *i* that needs the result of the load, we need to stall the pipeline for one cycle , that is
  - instruction *i-1* should progress normally
  - instruction *i* should not progress
  - no new instruction should be fetched
- The hazard detection unit should operate during the ID stage
- When processing instruction *i*, how do we know instruction *i-1* is a Load ?
  - Memread signal is asserted in ID/EX

Hazard detection unit (c'd)

- How do we know we should stall
  - instruction *i-1* is a Load and either
    - ID/EX rt = IF/ID rs, or
    - ID/EX rt = IF/ID rt
- How do we prevent instruction *i* to progress
  - Put 0's in all control fields of ID/EX (becomes a no-op)
  - Don't change the IF/ID field (have a control line be asserted at every cycle to write it unless we have to stall)
- How do we prevent fetching a new instruction
  - Have a control line asserted only when we want to write a new value in the PC

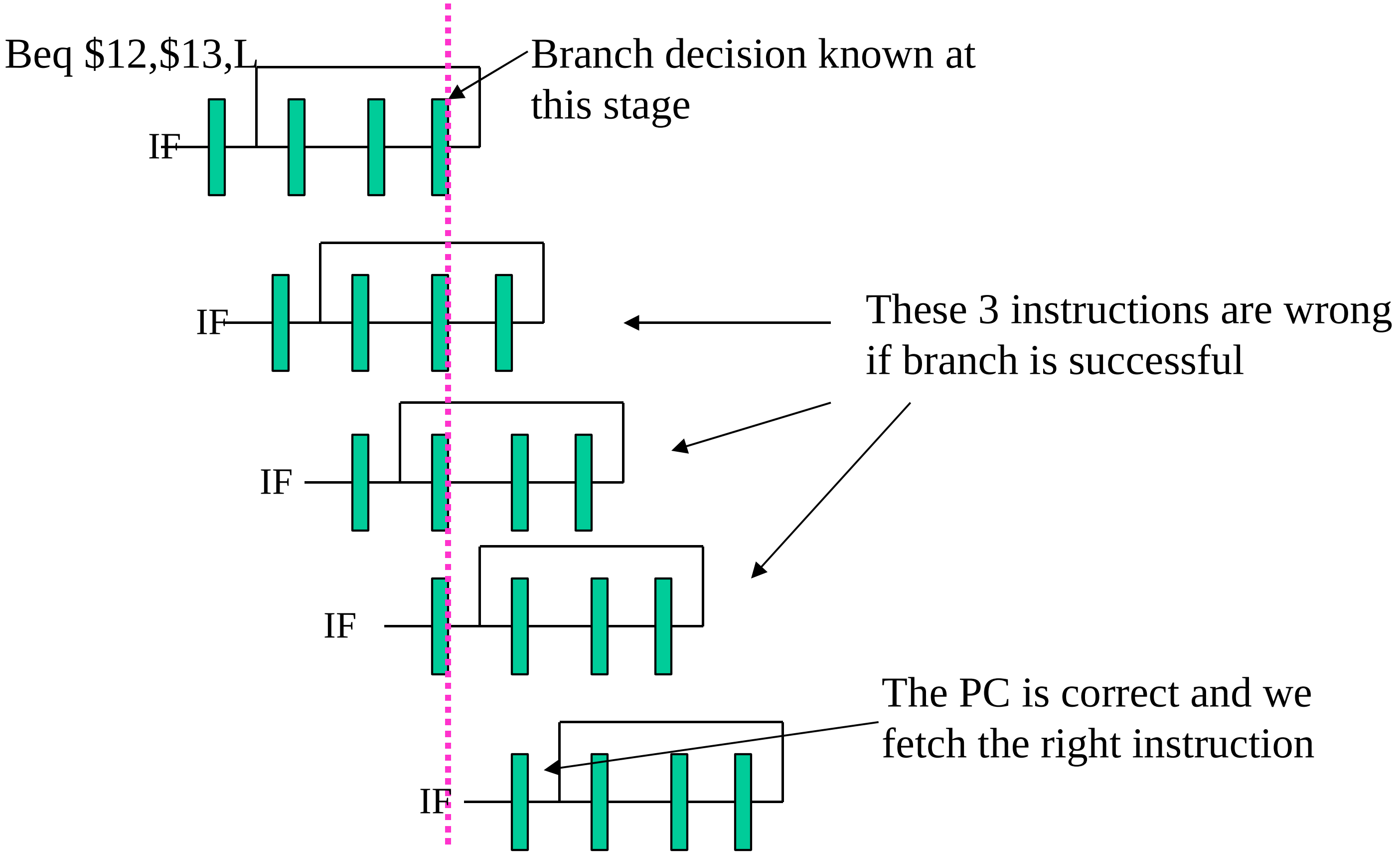
The (almost) overall picture for data hazards

- See Figure 6.46.
- What is missing
  - Forwarding when Load followed by a Store (mem to mem copy)
    - forwarding from MEM/WB stage to memory input
  - Details about immediate instructions, address computations and passing the contents of the store register from stage to stage (cf. Figure 6.43)

Control hazards

- Pipelining and branching don’t get along
- Transfer of control (jumps, procedure call/returns, successful branches) cause control hazards
- When a branch is known to succeed, at the Mem stage (but could be done one stage earlier), there are instructions in the pipeline in stages before Mem that
  - need to be converted into “no-op”
  - and we need to start fetching the correct instructions by using the right PC

Example of control hazard



Resolving control hazards

- Detecting a potential control hazard is easy
  - Look at the opcode
- We must insure that the state of the program is not changed until the outcome of the branch is known. Possibilities are:
  - Stall as soon as opcode is detected (cost 3 bubbles; same type of logic as for the load stall but for 3 cycles instead of 1)
  - Assume that branch won’t be taken (cost only if branch is taken; see next slides)
  - Use some predictive techniques

Assume branch not taken strategy

- We have a problem if branch is taken!
- “No-op” the “wrong” instructions
  - Once the new PC is known (in Mem stage)
    - Zero out the instruction field in the IF/ID pipeline register
    - For the instruction in the ID stage, use the signals that were set-up for data dependencies in the Load case
    - For the instruction in the EX stage, zero out the result of the ALU (e. g, make the result register be register $0)

Optimizations

- Move up the result of branch execution
  - Do target address computation in ID stage (like in multiple cycle implementation)
  - Comparing registers is “fast”; can be done in first phase of the clock and setting PC in the second phase.
  - Thus we can reduce stalling time by 1 bubble
- In the book, they reduce it by 2 bubbles but….
  - The organization as shown is slightly flawed (they forgot about extra complications in forwarding ….)

Branch prediction

- Instead of assuming “branch not taken” you can have a table keeping the history of past branches
  - We’ll see how to build such tables when we study caches
  - History can be restricted to 2-bit “saturating counters” such that it takes two wrong prediction outcomes before changing your prediction
  - If predicted taken, will need only 1 bubble since PC can be computed during ID stage.
  - There even exists schemes where you can predict and not lose any cycle on predicted taken, of course if the prediction is correct
- Note that if prediction is incorrect, you need to flush the pipe as before

Current trends in microprocessor design

- *Superscalar* processors
  - Several pipelines, e.g., integer pipeline(s), floating-point, load/store unit etc
  - Several instructions are fetched and decoded at once. They can be executed concurrently if there are no hazards
- *Out-of-order execution* (also called dynamically scheduled processors)
  - While some instructions are stalled because of dependencies or other causes (cache misses, see later), other instructions down he stream can still proceed.
  - However results must be stored in program order!

Current trends (ct’d)

- *Speculative execution*
  - Predict the outcome of branches and continue processing with (of course) a recovery mechanism.
  - Because branches occur so often, the branch prediction mechanisms have become very sophisticated
  - Still in the research stage, predict the outcome of instructions w/o executing them!
- *VLIW (or EPIC)* (Very Long Instruction Word)
  - In “pure VLIW”, each pipeline (functional unit) is assigned a task at every cycle. The compiler does it.
  - A little less ambitious: have compiler generate long instructions (e. g., using 3 pipes; cf. Intel IA-64 or Merced)