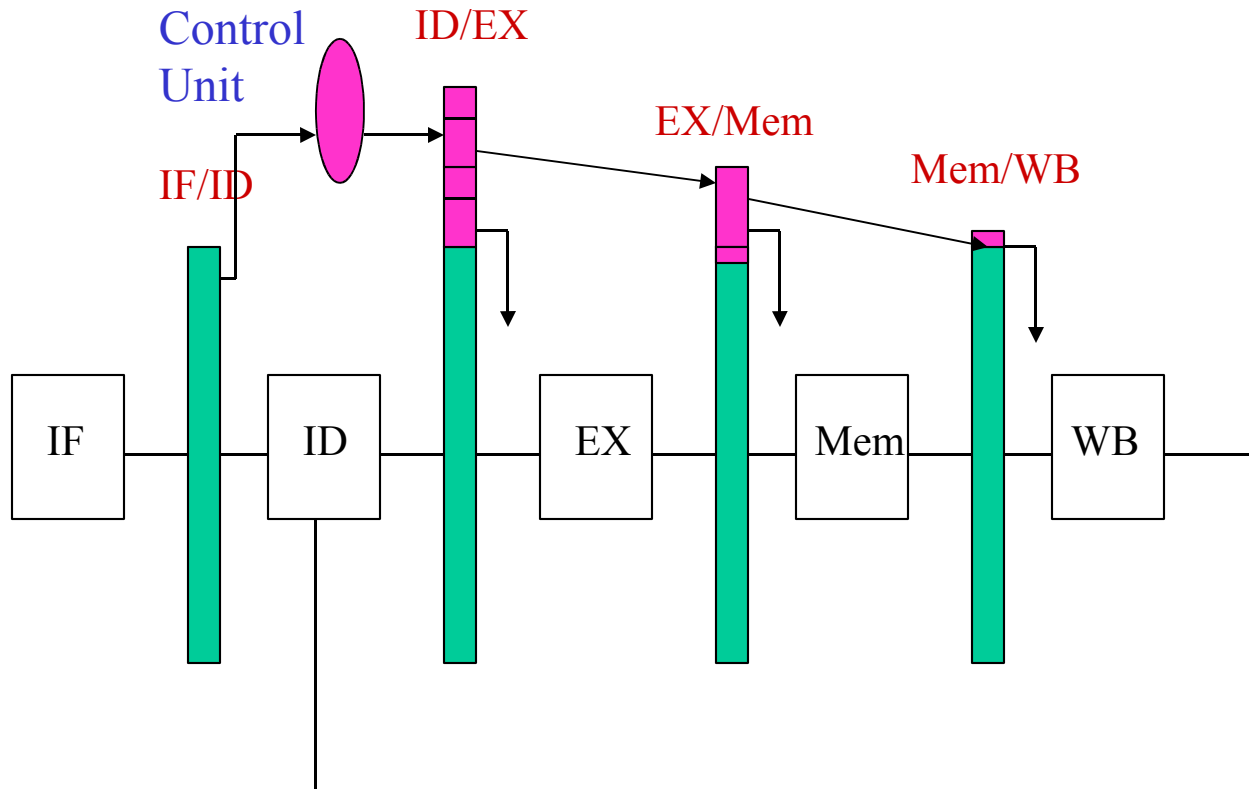


Pipeline Control unit (highly abstracted)



Where are the control signals needed?

- Very much like in multiple cycle implementation
 - Decompose in “states” that are now implicit in the sequence of pipeline registers
- Somewhat like single cycle implementation
 - All control line settings decided at decode stage
- Would be OK if pipeline were ideal but ...
- cf. Figure 6.22

Control (ideal case)

- Control signals are split among the 5 stages. For the ideal case no need for additional control (but just wait!)
- *Stage 1*: nothing special to control
 - read instr. memory and increment PC asserted at each cycle
- *Stage 2*: nothing. All instructions do the same
- *Stage 3*: Instruction dependent
 - Control signals for ALU sources and ALUop
 - Control signal for Regdest so the right name is passed along
- *Stage 4*: Control for memory (read/write) and for branches
- *Stage 5*: Control for source of what to write in the destination register

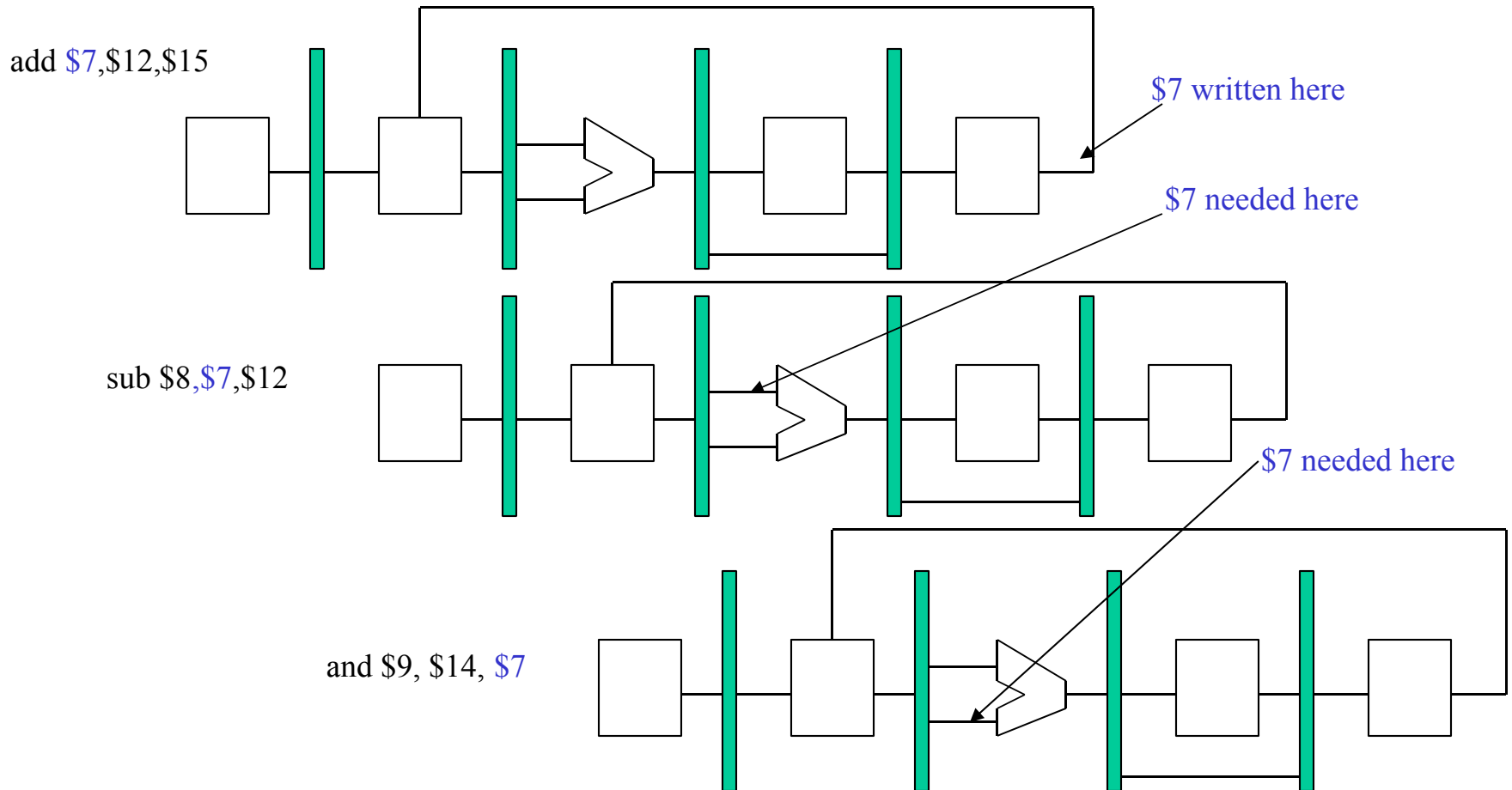
Hazards

- Recall
 - *structural* hazards: lack of resources (won't happen in our simple pipeline)
 - *data* hazards: due to dependencies between executing instructions
 - *control* hazards: flow of control is not sequential

Data dependencies

- The result of an operation is needed before it is stored back in the register file
 - Example:
add \$7, \$12, \$15 # put result in register 7
sub \$8, \$7, \$12 # use register 7 as a source
and \$9, \$14, \$7 # use register 7 as a source
 - The above dependence is called **RAW** (Read After Write)
 - Note that there is no dependency for register 12 which is used as a source in two operations
 - WAW (Write After Write) and WAR (Write After Read) dependencies can exist but not in our simple pipeline

Data dependencies in the pipe



Occurrences of data dependencies (detection)

- Data dependence (RAW) occurs when:
 - An instruction wants to read a register in stage 2, and
 - One instruction in either stage 3 or stage 4 is going to write that register
 - Note that if the instruction writing the register is in stage 5, this is fine since we can write a register and read it in the same cycle
- Data dependencies can occur between (not an exhaustive list):
 - Arithmetic instructions
 - A load and an arithmetic instruction needing the result of a load
 - An arithmetic instruction and a load/store (to compute the address)
 - An arithmetic instruction and a branch (to compare registers)

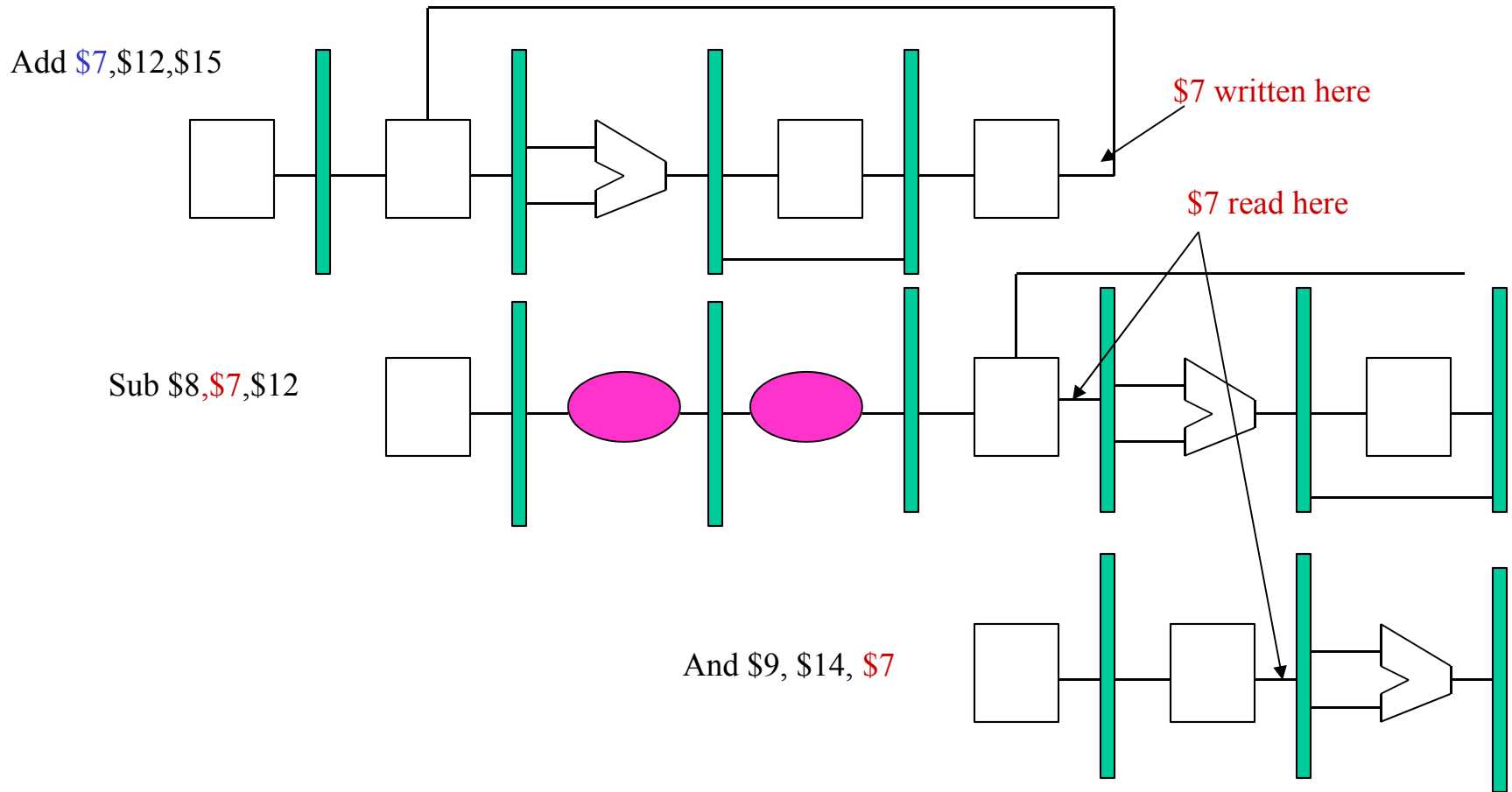
Resolving data dependencies (Potential solutions)

- There are several possibilities:
 - Have the compiler generate “no-ops”, i.e., instructions that do nothing while passing through the pipeline (original MIPS at Stanford; found to be too complex)
 - Stall the pipeline when the hardware detects the dependency, i.e., create *bubbles* (the resulting delays are the same as for no-ops)
 - Forwarding the result, generated in stage 3 or stage 4, to the appropriate input of the ALU. This is called *forwarding* or *bypassing*. Certainly more performance efficient at the cost of more hardware
 - In the case of a simple (unique) pipeline, cost is slightly more control and extra buses
 - If there were several pipelines, say n , communication grows as $O(n^2)$

Detection of data dependencies

- When an instruction reaches stage 2, the control unit will detect whether the names of the **result registers** of the **two previous instructions** match the name of the source registers for the current instruction.
 - Examples: EX/Mem write-register name = ID/EX rs
Mem/WB write-register name = ID/EX rt
etc ...

Example of stalls



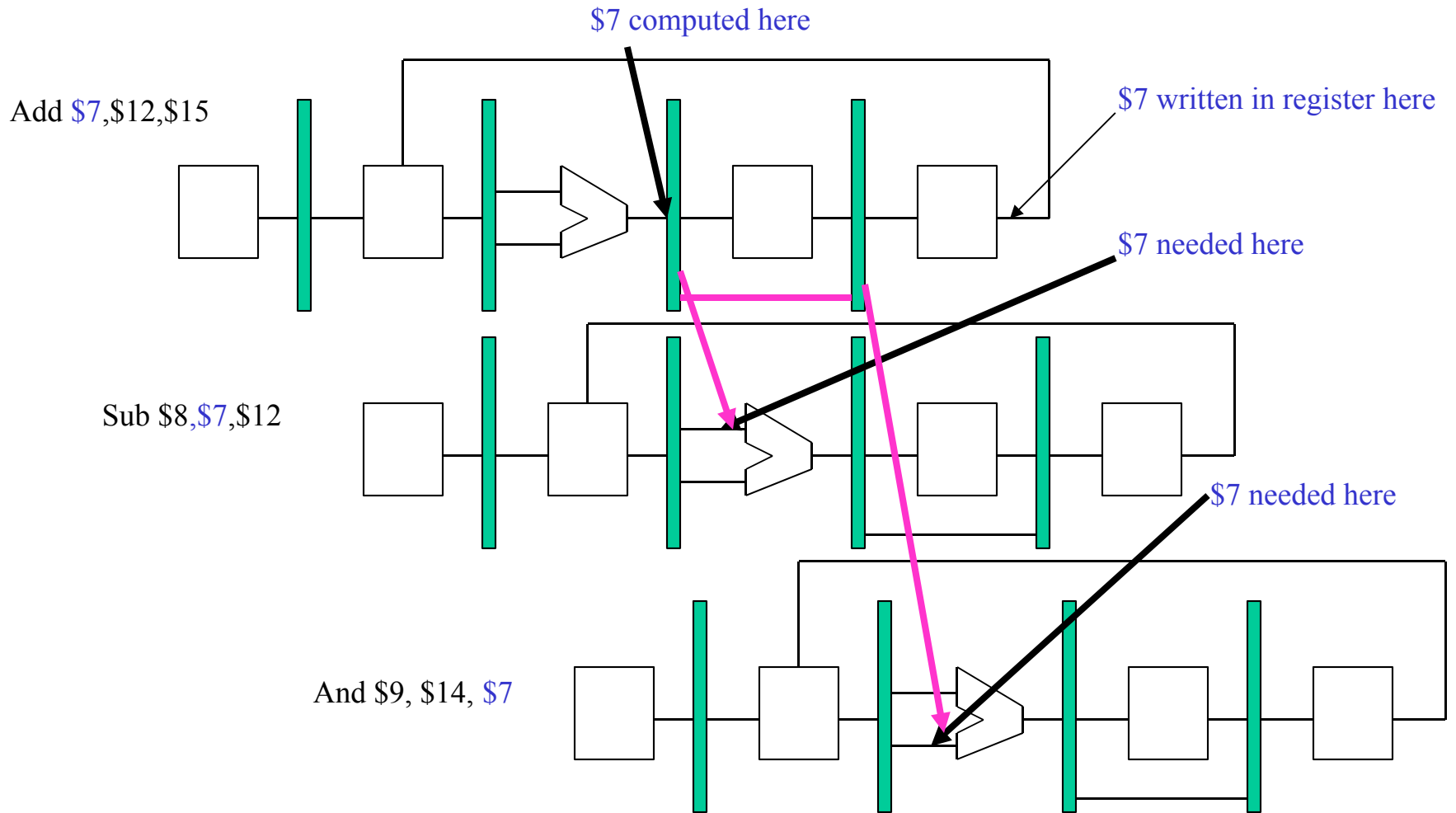
How to detect stalls and additional control

- Between instruction $i+1$ and instruction i (2 bubbles)
ID/EX write-register = IF/ID read-register 1 or IF/ID read-register 2
- Between instruction $i+2$ and instruction i (1 bubble)
EX/Mem write-register = IF/ID read-register 1 or IF/ID read-register 2
- Note that we are stalling an instruction in stage 2 (decode) thus
 - We must prevent fetching new instructions (otherwise PC and current instruction would be clobbered in IF/ID)
 - Requires control unit to create bubbles (set all control lines to 0 from stage 2 on) and prevent new instruction fetches

Forwarding

- Bubbles (or no-ops) are pessimistic since result is present before stage 5
 - In stage 3 for arithmetic instructions
 - In stage 4 for loads
- So why not forward directly the result from stage 3 (or 4) to the ALU
- Note that the state of the process (i.e., writing in registers) is still modified only in stage 5
 - The importance of this will become clear when we look at exceptions.

Forwarding in the pipe



Forwarding implementation

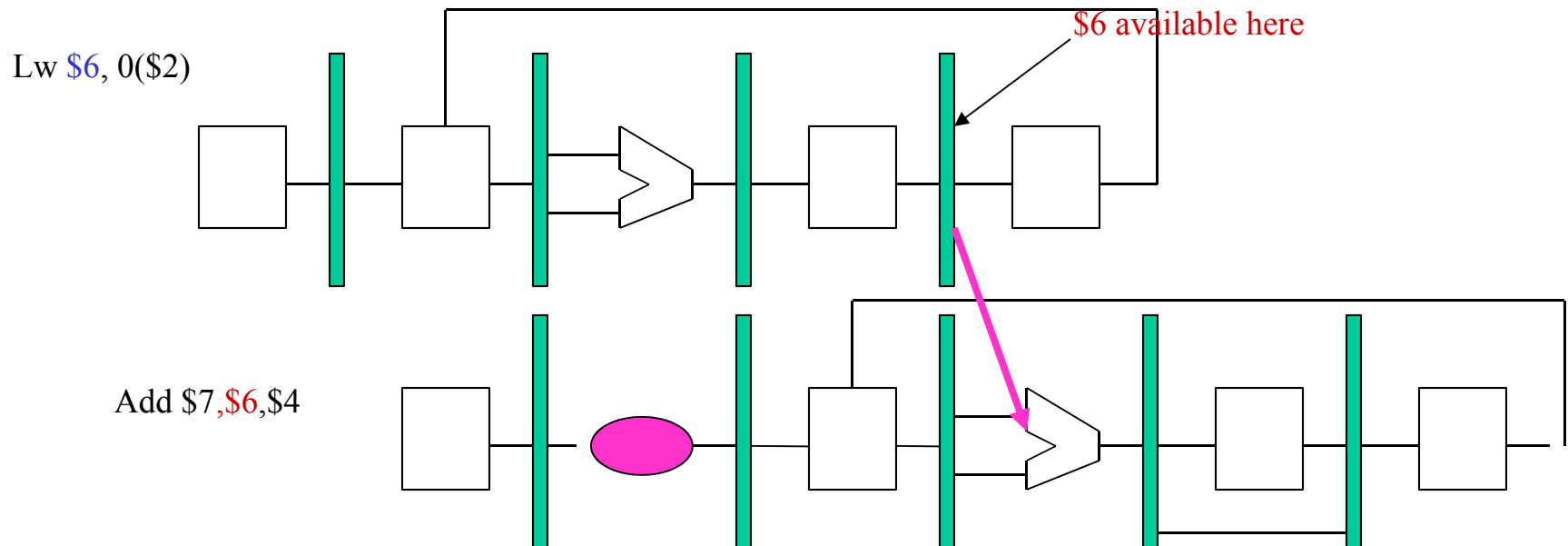
- Add busses to the data path so that inputs to ALU can be taken from
 - register file
 - EX/Mem pipeline register
 - Mem/WB pipeline register
- Have a “control forwarding unit” that detects
 - forwarding between instructions $i+1$ and i and between instructions $i+2$ and i (note that both can happen at the same time for the two sources)
- Expand muxes to allow these new choices

Still need for stalling

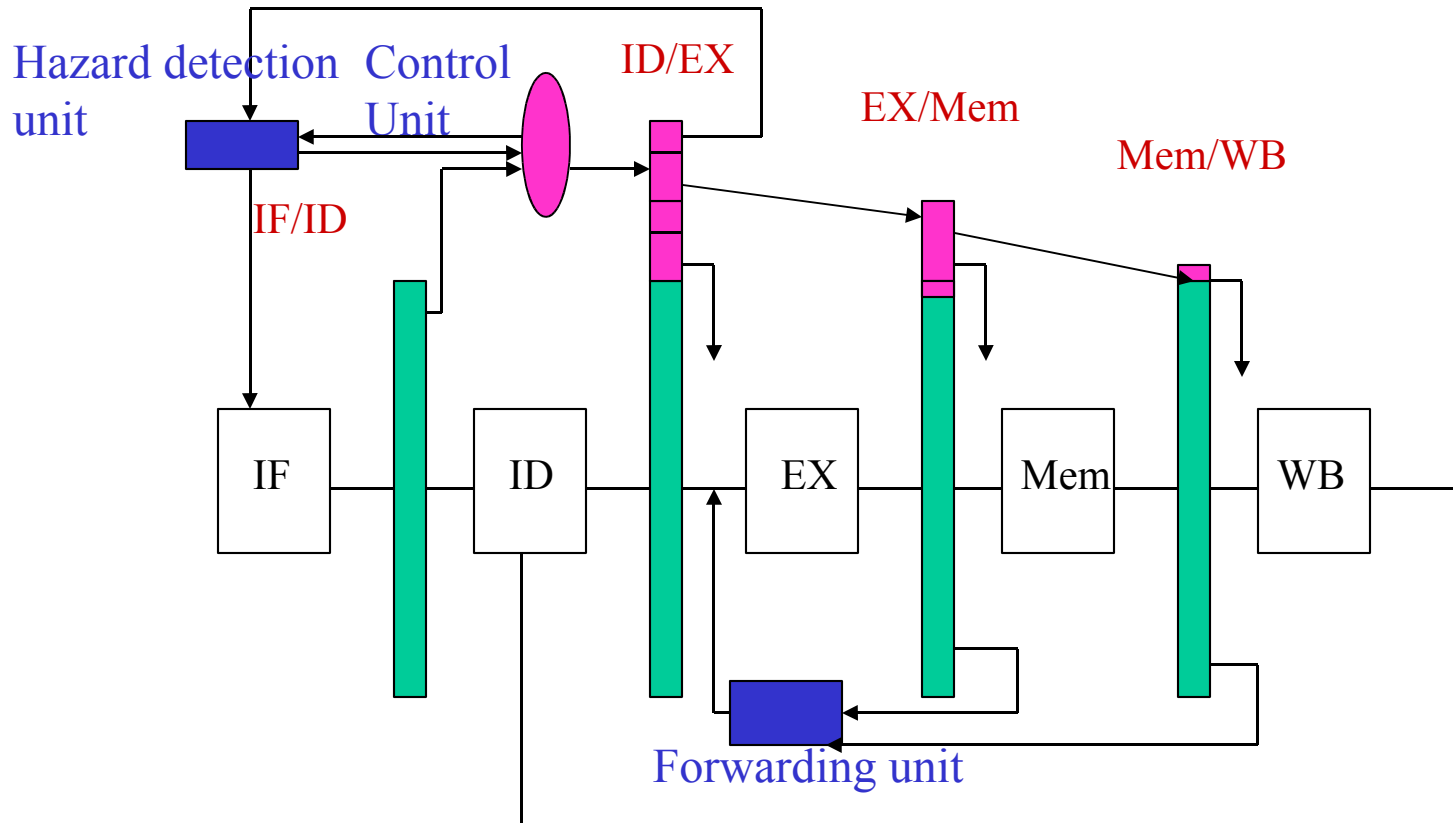
- Alas, we can't get rid of bubbles completely because the result of a load is only available at the end of stage 4
 - Example : `lw $6,0($2)`
`add $7,$6,$4`

We need to stall for 1 cycle and then forward

The Load stalling case



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 6.36.
- 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