Hazards

Pipelining as we have seen it up until now was perfect.

But, in real life we might not be able to fill the pipe fully because of hazards:

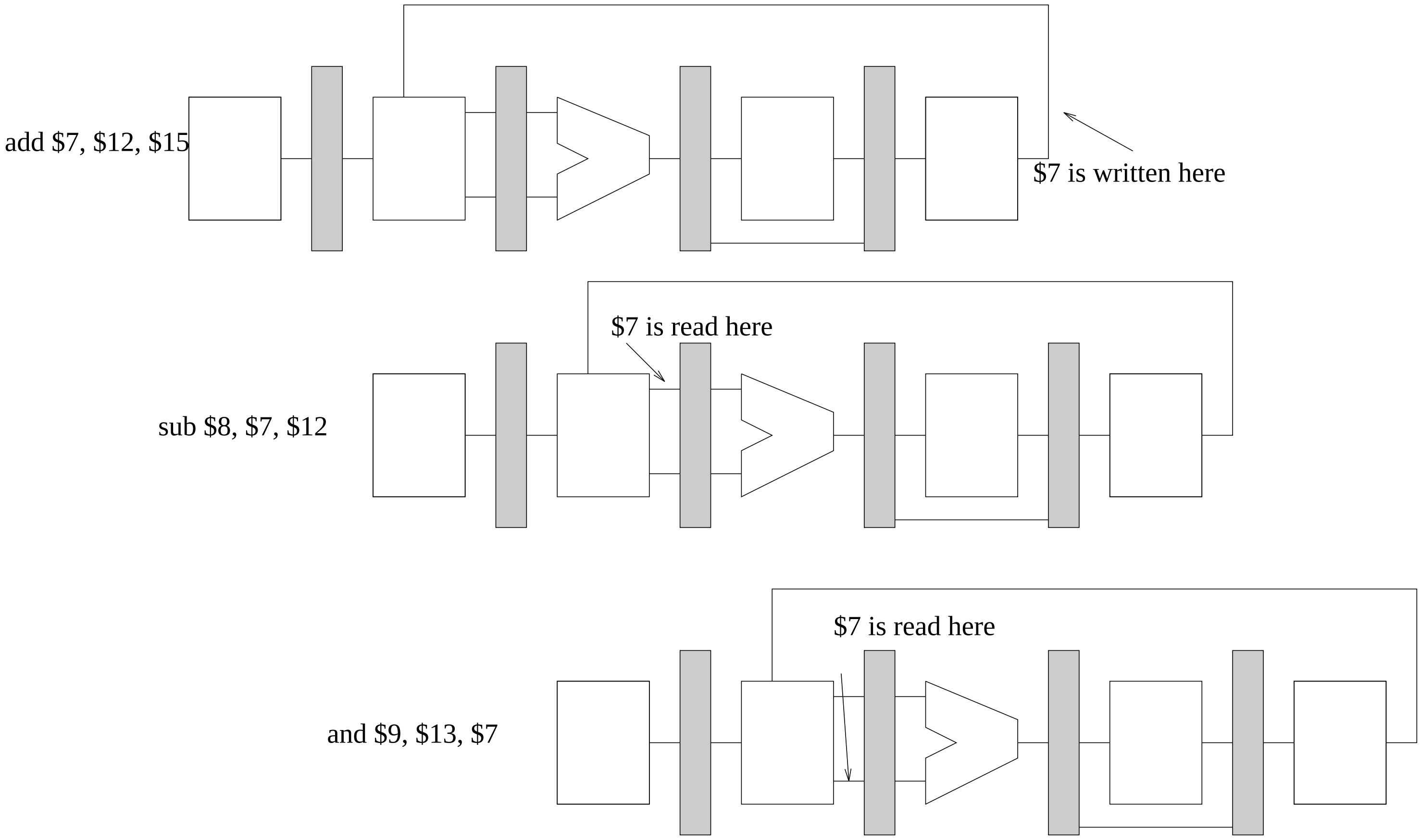
- data hazards or *dependencies*. For example the result of an operation is needed before it is computed

  ```
  add $7, $12, $15 #put result in register 7
  sub $8, $7, $12  #use register 7 as an operand
  and $9, $13, $7  #use register 7 as an operand
  ```

  Note that there is no dependency for register 12. It is used as a source in consecutive operations but there is no problem with that.

- control hazards or *branches*. If the branch is successful, then the instructions fetched after the branch in the pipe are the wrong ones

Data dependencies in the pipe



Detection of data dependencies

Data dependencies occur when an instruction tries to read a register in stage 2 (Decode and register fetch) and this register will be written by a previous instruction that has not yet reached stage 5 (Write back)

Notation: hazard because:

EX/MEM write-register = ID/EX read register 1 or 2

with of course other possible pipeline registers.

This can occur in arithmetic operations as well as in load operations.

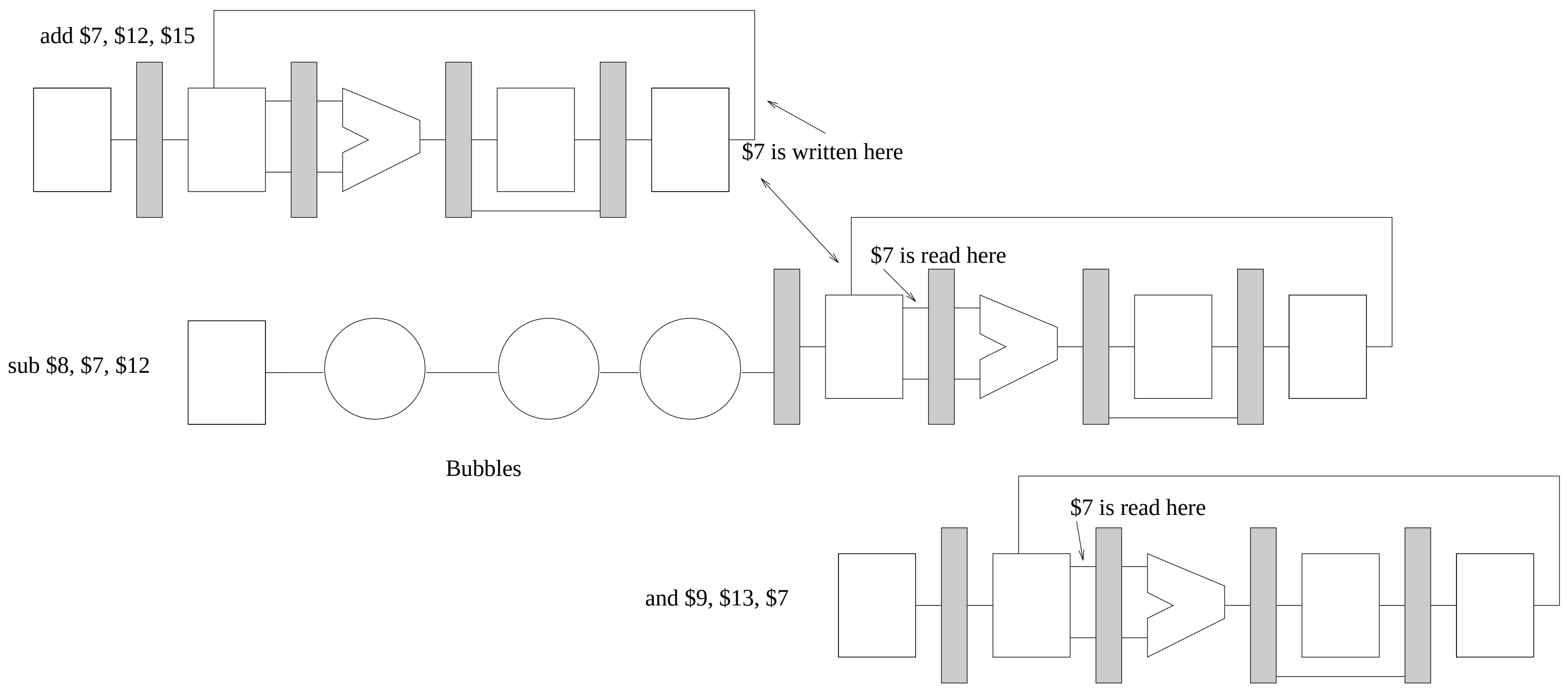
In modern microprocessors with several ALU's thus allowing more concurrency, and, e.g., floating-point units that take longer than integer units etc., other kinds of data hazards may occur.

Resolving data dependencies

There are several possibilities:

1. Have the compiler generate No-ops (i.e., an instruction that does nothing while passing through the pipeline)
2. Stall the pipeline when the hardware detects a dependency. This creates *bubbles* in the pipe.
3. Forwarding a result as an ALU source (see in a couple of slides)

Example of stalls



How to detect stalls and additional control

Between instruction i+1 and instruction i (3 bubbles):

- ID/EX write-register = IF/ID read-register 1 or 2
  (in fact this is slightly more complex because write-register can be rd or rt depending on the type of instruction)

Between instruction i+2 and instruction i (2 bubbles):

- EX/MEM write-register = IF/ID read-register 1 or 2

Between instruction i+3 and instruction i (1 bubble):

- MEM/WB write-register = IF/ID read-register 1 or 2

Note that when stalling, we are stalling an instruction in the Decode stage.

Therefore, we must prevent fetching new instructions (otherwise we would clobber the PC and the IF/ID register). So we need control lines to:

- Create bubbles. This can be done by setting all control signals that are passed from stage 2 on to 0, hence we will have a no-op.
- Prevent new instruction fetches. This should be done for as many cycles as there are bubbles.

Forwarding

Bubbles are inserted in a pessimistic way since the data that is written in the last stage is often computed before the last stage:

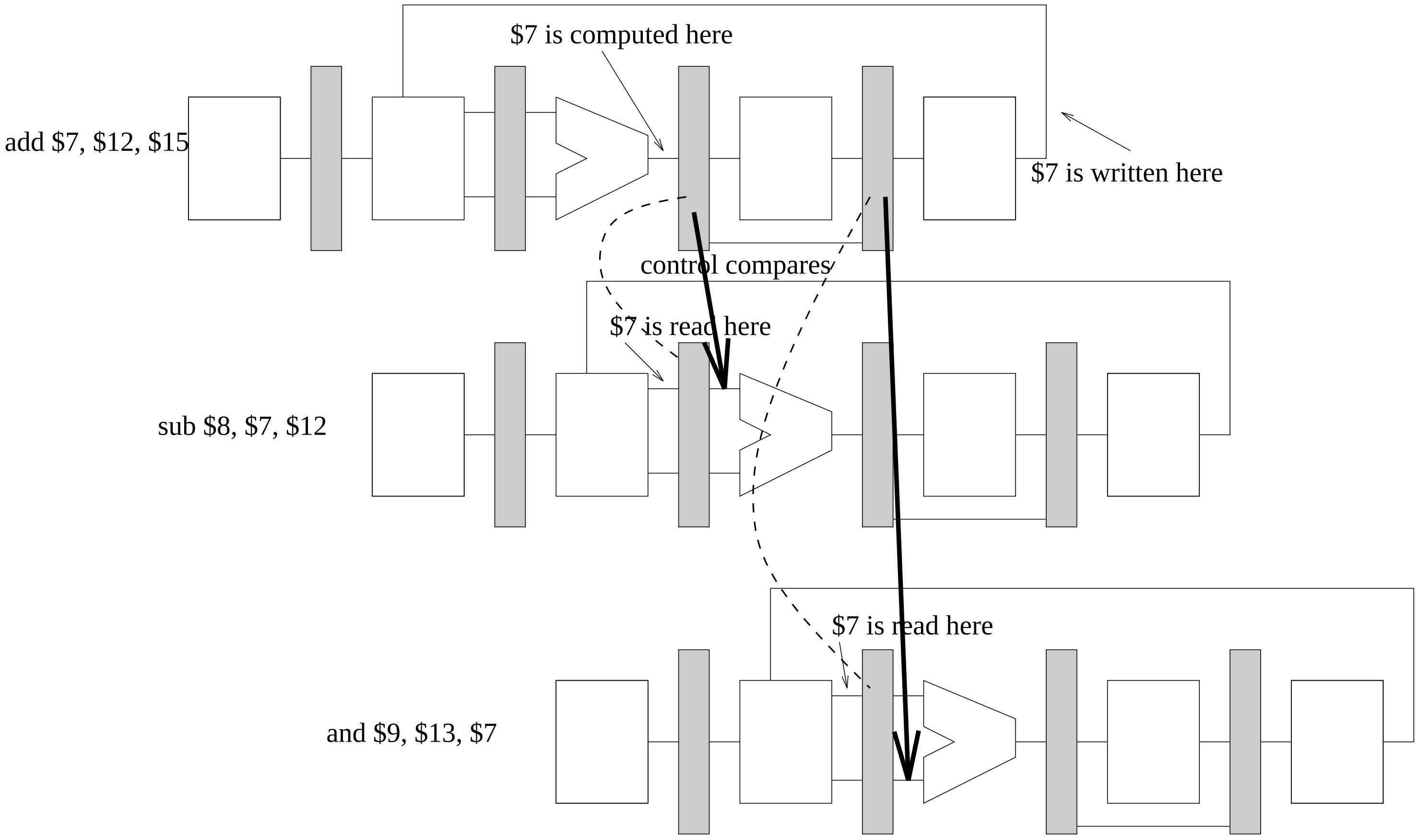
- at the end of the EXE stage for an arithmetic operation
- At the end of the MEM stage for a Load

So why not *forward* the result of a computation, or of a load, directly to the input of the ALU if it is needed in an instruction that would be stalled otherwise.

This is also often called *bypassing*.

Note that for reasons related to *interrupts* or *exceptions*, we do not want the *state* of the process (i.e., registers) to be modified until the last stage.

Forwarding example



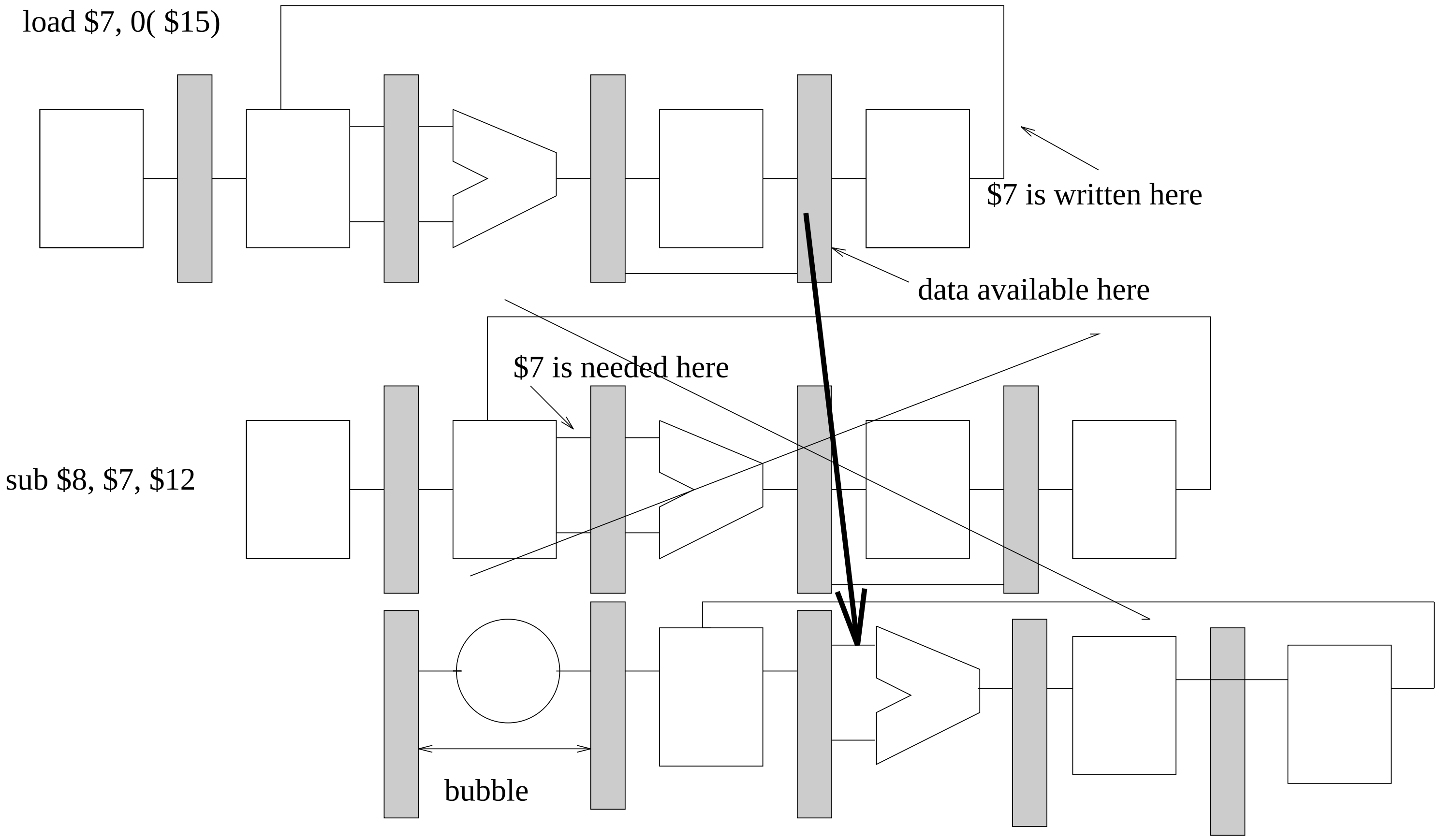
Forwarding implementation

Change the data path so that data can be taken from either EX/MEM or MEM/WB to be forwarded to one of the inputs of the ALU (this assumes that the writing of a register and putting it as input to the ALU can be done in one stage) Needs:

- Logic to detect the forwarding. This can be done:
  - at stage 3 of instruction i to forward to stage 2 of instruction $i + 1$
  - at stage 4 of instruction i to forward to stage 2 of instruction $i + 2$
- additional inputs to multiplexors to ALU sources (inputs can come from the register file, the EX/MEM register and the MEM/WB register)

Alas if we have a load followed by an arithmetic operation needing the result of the load, we still have to introduce a bubble (and we must therefore keep some logic for stalling).

Example of Load hazard



Control unit extension for data hazards and stalls

cf. Figure 6.46 in your book

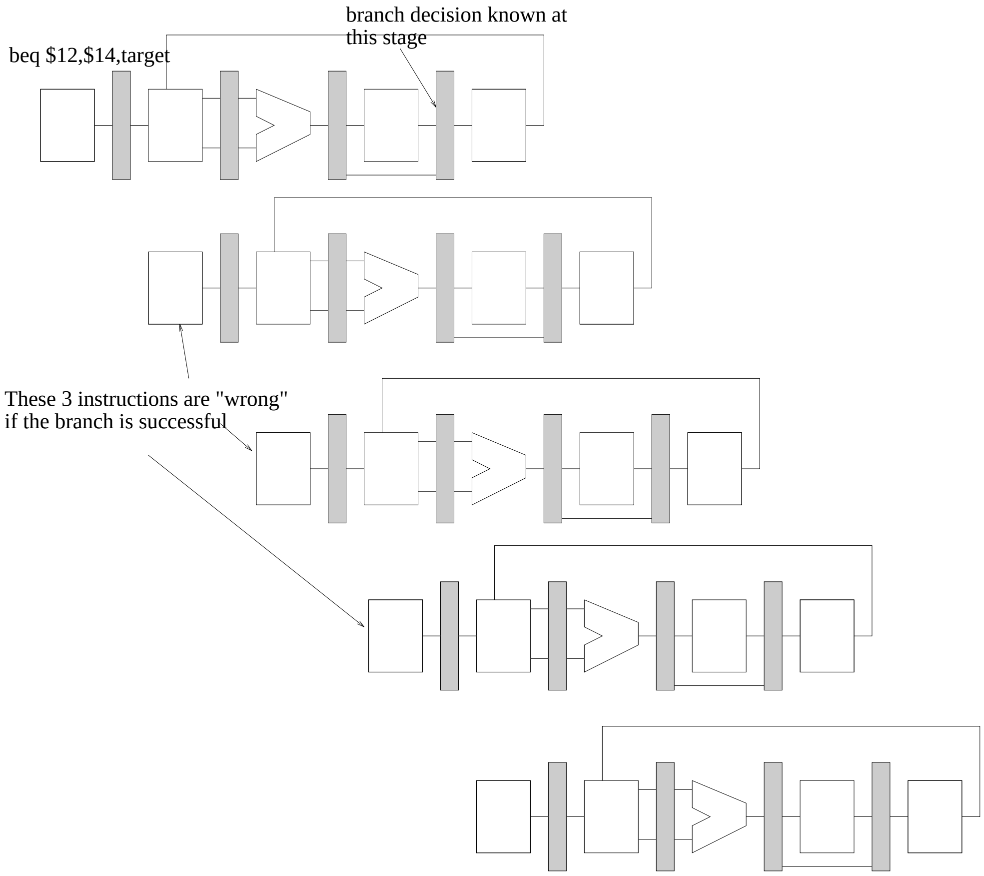
Control Hazards

Pipelining and branching don't get along

Transfer of control (jumps, procedure calls, returns, successful branches) cause control hazards.

When a branch is known to succeed (at the MEM stage), there are instructions already in the pipeline that are the *wrong* instructions.

Example of Control Hazard



Resolving control hazards

Detecting a *potential* control hazard is easy: look at the opcode.

Since the *wrong* instructions are in the pipeline, we must be sure that the *state* of the process is not changed until the outcome is determined.

Possibilities are:

1. Stall as soon as the branch opcode is detected (ID stage). Resume when the branch outcome is determined (costly, introduces 3 bubbles).
2. Assume that the branch will not be taken (cost is nothing if the branch fails).
3. Use other techniques (branch prediction)

In case 2, when branch is taken, we need to *flush* the pipeline.

- For the instruction in the IF stage, zero out the instruction field in IF/ID)
- For the instruction in the ID stage, use the signals for stalls that were set up for data dependencies (w/o forwarding)
- For the instruction in the EX stage, use new multiplexors to zero out results out of the ALU.

Optimizations

Move up the result of the branch execution. Recall the multiple cycle implementation where the target address was computed in cycle 2. Could be done here and would “save” 1 bubble in the case of a taken branch.

Use *branch prediction*. Have a *branch prediction table* (also called branch prediction buffer, or branch table buffer) that records whether the branch should be taken or not, based on the past history of execution of that branch (in general use a 2-bit saturating counter scheme such that you need 2 wrong predictions before changing your prediction). If predicted taken, the PC can be computed during the ID stage and only 1 bubble will have to be inserted.

There are even ways to lose no bubble at all but ... this is outside the scope of these notes.

Note that if the prediction is wrong, you still need to flush the pipe and recover from the right next instruction address!

Some modern trends

Superscalar processors have several pipelines. For example, 1 for arithmetic/logical on integers, one for floating-point, one for load/store etc.

Several instructions are fetched at once. The decoder decides whether consecutive instructions are conflicting or not (because of structural –same resources– or data dependences). If not they can be executed "at the same time".

Out-of-order execution (also called dynamic pipelining). While some instructions are stalled (e.g., if an Add follows a Load with a data dependence), maybe there are instructions down the stream that could be executed. This occurs fairly often if you have multiple pipes of varying number of stages. Although instructions are executed out-of-order, their results are finalized in order.

Alas, we do not have the time to go any further on these topics.