4.7 Data hazards: Forwarding versus stalling

> "What do you mean, why's it got to be built? It's a bypass. You've got to build bypasses."
> *Douglas Adams, The Hitchhiker's Guide to the Galaxy, 1979.*

The examples in COD Section 4.6 (Pipelined datapath and control) show the power of pipelined execution and how the hardware performs the task. It's now time to take off the rose-colored glasses and look at what happens with real programs. The LEGv8 instructions in COD Figures 4.42 (Multiple-clock-cycle pipeline diagram ...), 4.43 (Traditional multiple-clock-cycle pipeline diagram ...), and 4.44 (The single-clock-cycle diagram corresponding to clock cycle 5 ...) were independent; none of them used the results calculated by any of the others. Yet, in COD Section 4.5 (An overview of pipelining), we saw that data hazards are obstacles to pipelined execution.

Let's look at a sequence with many dependences, shown in color:

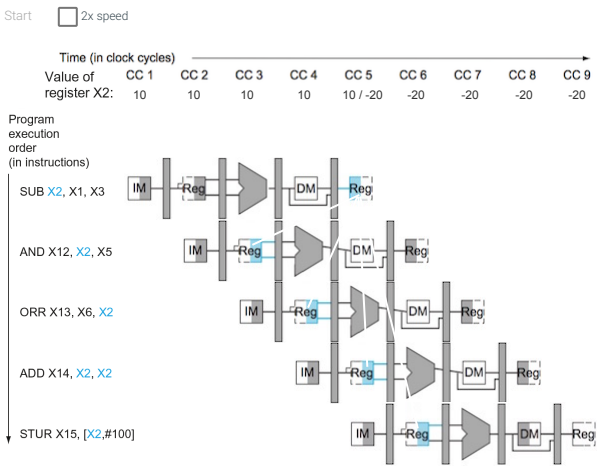
```
SUB   X2, X1,X3     // Register X2 written by SUB
AND   X12,X2,X5     // 1st operand(X2) depends on SUB
OR    X13,X6,X2     // 2nd operand(X2) depends on SUB
ADD   X14,X2,X2     // 1st(X2) & 2nd(X2) depend on SUB
STUR  X15,[X2,#100] // Base (X2) depends on SUB
```

The last four instructions are all dependent on the result in register **X2** of the first instruction. If register **X2** had the value 10 before the subtract instruction and -20 afterwards, the programmer intends that -20 will be used in the following instructions that refer to register **X2**.

How would this sequence perform with our pipeline? The animation below illustrates the execution of these instructions using a multiple-clock-cycle pipeline representation. To demonstrate the execution of this instruction sequence in our current pipeline, the top of the animation below shows the value of register **X2**, which changes during the middle of clock cycle 5, when the **SUB** instruction writes its result.

All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into **X2**, and all the following instructions read **X2**. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards*.

PARTICIPATION ACTIVITY 4.7.1: Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences (COD Figure 4.51).



The last potential hazard can be resolved by the design of the register file hardware: What happens when a register is read and written in the same clock cycle? We assume that the write is in the first half of the clock cycle and the read is in the second half, so the read delivers what is written. As is the case for many implementations of register files, we have no data hazard in this case.

The animation above shows that the values read for register **X2** would *not* be the result of the **SUB** instruction unless the read occurred during clock cycle 5 or later. Thus, the instructions that would get the correct value of -20 are **ADD** and **STUR**; the **AND** and **ORR** instructions would get the incorrect value 10! Using this style of drawing, such problems become apparent when a dependence line goes backward in time.

PARTICIPATION ACTIVITY 4.7.2: Pipeline data hazards.

Consider the above figure showing pipelined dependences in a five-instruction sequence.

1) Each blue line represents a data dependence.
   - ○ True
   - ○ False

2) Each blue line represents a pipeline data hazard.
   - ○ True
   - ○ False

3) The STUR instruction would read the

old (wrong) value from X2.

- ○ True
- ○ False

4) The ORR instruction would read the old (wrong) value from X2.

- ○ True
- ○ False

5) Suppose a sixth instruction was `SUB X3, X13, X14`. A pipeline data hazard would exist.

- ○ True
- ○ False

6) Suppose the second instruction was instead: `ORR X13, X6, X5`. A pipeline data hazard would exist.

- ○ True
- ○ False

As mentioned in COD Section 4.5 (An overview of pipelining), the desired result is available at the end of the EX stage of the `SUB` instruction or clock cycle 3. When are the data actually needed by the `AND` and `ORR` instructions? The answer is at the beginning of the EX stage of the `AND` and `ORR` instructions, or clock cycles 4 and 5, respectively. Thus, we can execute this segment without stalls if we simply *forward* the data as soon as it is available to any units that need it before it is ready to read from the register file.

How does forwarding work? For simplicity in the rest of this section, we consider only the challenge of forwarding to an operation in the EX stage, which may be either an ALU operation or an effective address calculation. This means that when an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, we actually need the values as inputs to the ALU.

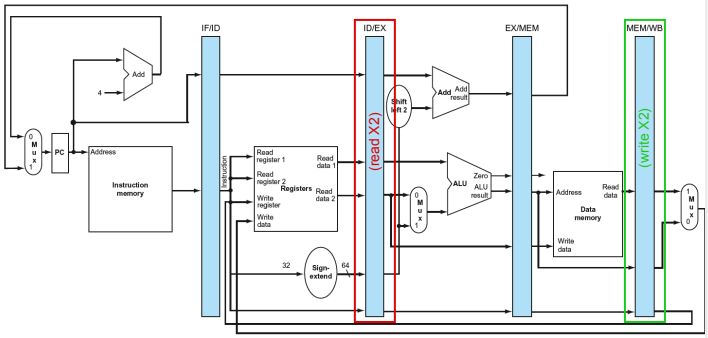
A notation that names the fields of the pipeline registers allows for a more precise notation of dependences. For example, "ID/EX.RegisterRn1" refers to the number of one register whose value is found in the pipeline register ID/EX; that is, the one from the first read port of the register file. The first part of the name, to the left of the period, is the name of the pipeline register; the second part is the name of the field in that register. Using this notation, the two pairs of hazard conditions are

- 1a. EX/MEM.RegisterRd = ID/EX.RegisterRn1
- 1b. EX/MEM.RegisterRd = ID/EX.RegisterRm2
- 2a. MEM/WB.RegisterRd = ID/EX.RegisterRn1
- 2b. MEM/WB.RegisterRd = ID/EX.RegisterRm2

PARTICIPATION ACTIVITY 4.7.3: Forwarding data.

Start ☐ 2x speed

1a. EX/MEM.RegisterRd = ID/EX.RegisterRn1 2a. MEM/WB.RegisterRd = ID/EX.RegisterRn
1b. EX/MEM.RegisterRd = ID/EX.RegisterRm2 2b. MEM/WB.RegisterRd = ID/EX.RegisterRn



The first hazard in the earlier-shown instruction sequence (`SUB`, `AND`, `ORR`, `ADD`, `STUR`) is on register `X2`, between the result of `SUB X2, X1, X3` and the first read operand of `AND X12, X2, X5`. This hazard can be detected when the `AND` instruction is in the EX stage and the prior instruction is in the MEM stage, so this is hazard 1a:

EX/MEM.RegisterRd = ID/EX.RegisterRn1 = `X2`

Example 4.7.1: Dependence detection.

Classify the dependences in this sequence:

```
SUB  X2,   X1, X3     // Register X2 set by SUB
AND  X12,  X2, X5     // 1st operand(X2) set by SUB
OR   X13,  X6, X2     // 2nd operand(X2) set by SUB
ADD  X14,  X2, X2     // 1st(X2) & 2nd(X2) set by SUB
STUR X15, [X2,#100]   // Index(X2) set by SUB
```

Answer

As mentioned above, the `SUB–AND` is a type 1a hazard. The remaining hazards are as follows:

- The `SUB–ORR` is a type 2b hazard:

  MEM/WB.RegisterRd = ID/EX.RegisterRm2 = `X2`
- The two dependences on `SUB–ADD` are not hazards because the register file supplies the proper data during the ID stage of `ADD`.
- There is no data hazard between `SUB` and `STUR` because `STUR` reads `X2` the clock cycle *after* `SUB` writes `X2`.

PARTICIPATION ACTIVITY 4.7.4: Identifying hazards.

Associate the hazard type with the second instruction of an instruction pair causing a hazard. Move "Start" to the first instruction.

2a. MEM/WB.RegisterRd = ID/EX.RegisterRn1 **Start**

1a. EX/MEM.RegisterRd = ID/EX.RegisterRn1

1b. EX/MEM.RegisterRd = ID/EX.RegisterRm2 **No hazard**

ADD X1, X2, X3

ADD X4, X5, X1

ADD X6, X1, X2

ADD X8, X9, X1

ADD X10, X8, X11

Reset

Because some instructions do not write registers, this policy is inaccurate; sometimes it would forward when it shouldn't. One solution is simply to check to see if the RegWrite signal will be active: examining the WB control field of the pipeline register during the EX and MEM stages determines whether RegWrite is asserted. Recall that LEGv8 requires that every use of `XZR(X31)` as an operand must yield an operand value of 0. If an instruction in the pipeline has `XZR` as its destination (for example, `SUBS XZR, X1, 2`), we want to avoid forwarding its possibly nonzero result value. Not forwarding results destined for `XZR` frees the assembly programmer and the compiler of any requirement to avoid using `XZR` as a destination. The conditions above thus work properly as long we add EX/MEM. RegisterRd ≠ 31 to the first hazard condition and MEM/WB.RegisterRd ≠ 31 to the second.

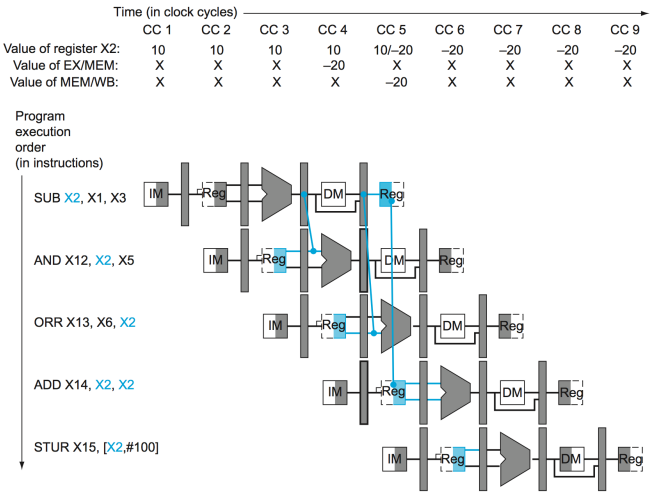
The last detail is to recall that two instructions—`STUR` and `CBZ`—use Rt (bits 4:0) to specify the second register operand instead of the Rm field (bits 20:16). The pipeline resolves this decision in the ID stage using a 2:1 multiplexor and the Reg2Loc control signal. We can simplify pipeline control by using the output of this multiplexor instead of the actual Rm field so that control always sees the proper second register number. We'll still call the number RegisterRm, but remember it handles `STUR` and `CBZ` correctly too.

Now that we can detect hazards, half of the problem is resolved—but we must still forward the proper data.

The figure below shows the dependences between the pipeline registers and the inputs to the ALU for the same code sequence as in the animation above (COD Figure 4.51). The change is that the dependence begins from a *pipeline* register, rather than waiting for the WB stage to write the register file. Thus, the required data exist in time for later instructions, with the pipeline registers holding the data to be forwarded.

Figure 4.7.1: The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the AND instruction and ORR instruction by forwarding the results found in the pipeline registers (COD Figure 4.52).

The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the `ADD` does not stall, but the values come from the register file instead of a pipeline register. Register file "forwarding"—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register `X2` having the value 10 at the beginning and -20 at the end of the clock cycle.

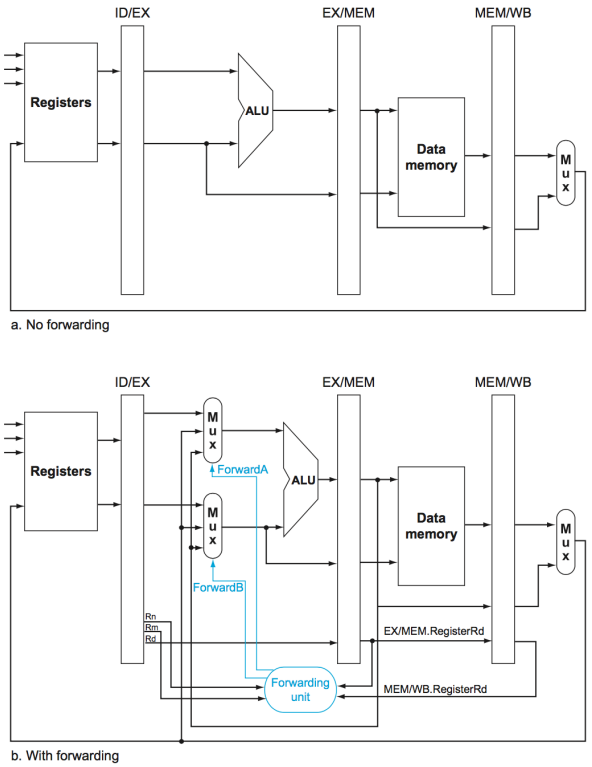


If we can take the inputs to the ALU from *any* pipeline register rather than just ID/EX, then we can forward the correct data. By adding multiplexors to the input of the ALU, and with the proper controls, we can run the pipeline at full speed in the presence of these data hazards.

For now, we will assume the only instructions we need to forward are the four R-format instructions: `ADD`, `SUB`, `AND`, and `ORR`. The figure below shows a close-up of the ALU and pipeline register before and after adding forwarding. COD Figure 4.54 (The control values for the forwarding multiplexors ...) shows the values of the control lines for the ALU multiplexors that select either the register file values or one of the forwarded values.

Figure 4.7.2: On the top are the ALU and pipeline registers before adding forwarding (COD Figure 4.53).

On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware.



PARTICIPATION ACTIVITY 4.7.5: A forwarding example.

Start 2x speed

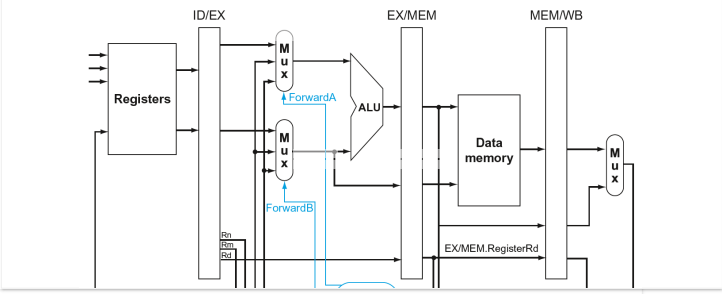


Figure 4.7.3: The control values for the forwarding multiplexors in the figure above (COD Figure 4.54).

The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

This forwarding control will be in the EX stage, because the ALU forwarding multiplexors are found in that stage. Thus, we must pass the operand register numbers from the ID stage via the ID/EX pipeline register to determine whether to forward values. Before forwarding, the ID/EX register had no need to include space to hold the Rn and Rm fields. Hence, they were added to ID/EX.

Let's now write both the conditions for detecting hazards, and the control signals to resolve them:

1. *EX hazard:*

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 31)
and (EX/MEM.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 31)
and (EX/MEM.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 10
```

This case forwards the result from the previous instruction to either input of the ALU. If the previous instruction is going to write to the register file, and the write register number matches the read register number of ALU inputs A or B, provided it is not register 31, then steer the multiplexor to pick the value instead from the pipeline register EX/MEM.

2. *MEM hazard:*

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  31)
and (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  31)
and (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 01
```

As mentioned above, there is no hazard in the WB stage, because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file.

One complication is potential data hazards between the result of the instruction in the WB stage, the result of the instruction in the MEM stage, and the source operand of the instruction in the ALU stage. For example, when summing a vector of numbers in a single register, a sequence of instructions will all read and write to the same register:

```
ADD X1,X1,X2
ADD X1,X1,X3
ADD X1,X1,X4
. . .
```

In this case, the result should be forwarded from the MEM stage because the result in the MEM stage is the more recent result. Thus, the control for the MEM hazard would be (with the additions highlighted):

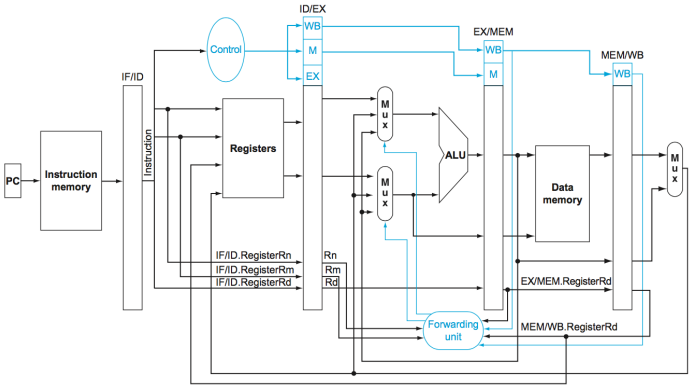
```
if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 31)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31)
         and (EX/MEM.RegisterRd = ID/EX.RegisterRn1))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 01

if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 31)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31)
         and (EX/MEM.RegisterRd = ID/EX.RegisterRm2))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 01
```

The figure below shows the hardware necessary to support forwarding for operations that use results during the EX stage. Note that the EX/MEM.RegisterRd field is the register destination for either an ALU instruction (which comes from the Rd field of the instruction) or a load (which comes from the Rt field, but we'll use the notation Rd in this section).

Figure 4.7.4: The datapath modified to resolve hazards via forwarding (COD Figure 4.55).

Compared with the datapath in COD Figure 4.50 (The corrected pipelined datapath ...), the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath, such as the branch hardware and the sign extension hardware.



If you would like to see more illustrated examples using single-cycle pipeline drawings, COD Section 4.13 (Advanced topic: An introduction to digital design using a hardware design language to describe and model a pipeline and more pipelining illustrations) has figures that show two pieces of LEGv8 code with hazards that cause forwarding.

Elaboration

Forwarding can also help with hazards when store instructions are dependent on other instructions. Since they use just one data value during the MEM stage, forwarding is easy. However, consider loads immediately followed by stores, useful when performing memory-to-memory copies in the LEGv8 architecture. Since copies are frequent, we need to add more forwarding hardware to make them run faster. If we were to redraw COD Figure 4.52 (The dependences between the pipeline registers move forward in time ...), replacing the `SUB` *and* `AND` *instructions with* `LDUR` *and* `STUR`*, we would see that it is possible to avoid a stall, since the data exist in the MEM/WB register of a load instruction in time for its use in the MEM stage of a store instruction. We would need to add forwarding into the memory access stage for this option. We leave this modification as an exercise to the reader.*

In addition, the signed-immediate input to the ALU, needed by loads and stores, is missing from the datapath in the figure above. Since central control decides between register and immediate, and since the forwarding unit chooses the pipeline register for a register input to the ALU, the easiest solution is to add a 2:1 multiplexor that chooses between the ForwardB multiplexor output and the signed immediate. The figure below shows this addition.

Figure 4.7.5: A close-up of the datapath in COD Figure 4.53 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input (COD Figure 4.56).

PARTICIPATION ACTIVITY 4.7.6: Forwarding unit.

Refer above to the conditions for detecting hazards.

1) Which is NOT a condition for setting the the ForwardA mux select lines to 10, causing forwarding of the ALU result in EX/MEM directly to the ALU's top input?
   - ○ EX/MEM.RegWrite
   - ○ ID/EX.RegWrite
   - ○ EX/MEM.RegisterRd != 0
   - ○ EX/MEM.RegisterRd = ID/EX.RegisterRn1

2) If the forwarding unit sets ForwardA to 10, the ALU's top input comes from ____.
   - ○ ID/EX
   - ○ EX/MEM
   - ○ MEM/WB

3) If the forwarding unit sets ForwardA to 01, the ALU's top input comes from ____.
   - ○ ID/EX
   - ○ EX/MEM
   - ○ MEM/WB

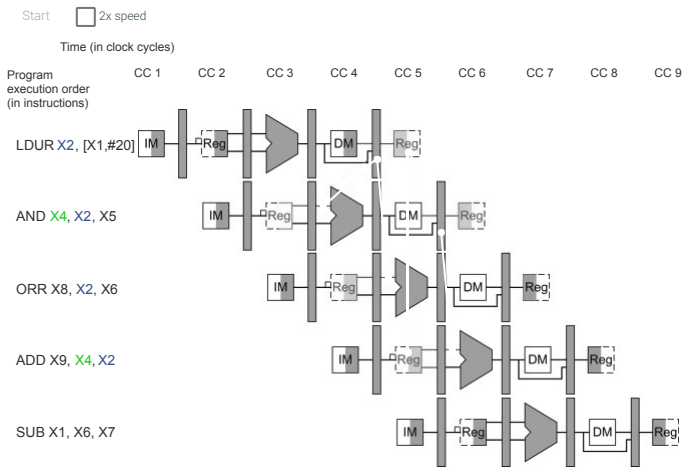
4) The forwarding unit sets ForwardA to 01 for which type of hazard?
   - ○ EX hazard
   - ○ MEM hazard

> If at first you don't succeed, redefine success.
> *Anonymous*

Data hazards and stalls

As we said in COD Section 4.5 (An overview of pipelining), one case where forwarding cannot save the day is when an instruction tries to read a register following a load instruction that writes the same register. The animation below illustrates the problem. The data are still being read from memory in clock cycle 4 while the ALU is performing the operation for the following instruction. Something must stall the pipeline for the combination of load followed by an instruction that reads its result.

PARTICIPATION ACTIVITY 4.7.7: A pipelined sequence of instructions (COD Figure 4.57).



Hence, in addition to a forwarding unit, we need a *hazard detection unit*. It operates during the ID stage so that it can insert the stall between the load and the instruction dependent on it. Checking for load instructions, the control for the hazard detection unit is this single condition:

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRd = IF/ID.RegisterRn1) or
        (ID/EX.RegisterRd = IF/ID.RegisterRm2)))
          stall the pipeline
```

Recall that we are using the RegisterRd to refer the register specified in instruction bits 4:0 for both load and R-type instructions. The first line tests to see if the instruction is a load: the only instruction that reads data memory is a load. The next two lines check to see if the destination register field of the load in the EX stage matches either source register of the instruction in the ID stage. If the condition holds, the instruction stalls one clock cycle. After this one-cycle stall, the forwarding logic can handle the dependence and execution proceeds. (If there were no forwarding, then the instructions in the figure above would need another stall cycle.)

If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, we would lose the fetched instruction. Preventing these two instructions from making progress is accomplished simply by preventing the PC register and the IF/ID pipeline register from changing. Provided these registers are preserved, the instruction in the IF stage will continue to be read using the same PC, and the registers in the ID stage will continue to be read using the same instruction fields in the IF/ID pipeline register. Returning to our favorite analogy, it's as if you restart the washer with the same clothes and let the dryer continue tumbling empty. Of course, like the dryer, the back half of the pipeline starting with the EX stage must be doing something; what it is doing is executing instructions that have no effect: *nops*.

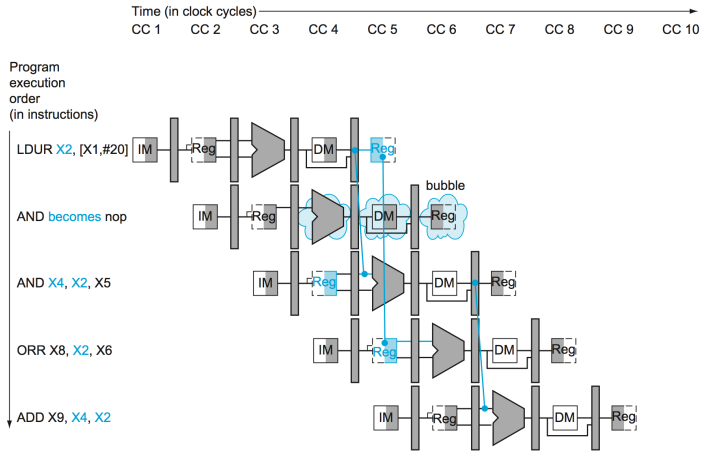
nop: An instruction that does no operation to change state.

How can we insert these nops, which act like bubbles, into the pipeline? In COD Figure 4.48 (The values of the control lines are the same …), we see that deasserting all eight control signals (setting them to 0) in the EX, MEM, and WB stages will create a "do nothing" or nop instruction. By identifying the hazard in the ID stage, we can insert a bubble into the pipeline by changing the EX, MEM, and WB control fields of the ID/EX pipeline register to 0. These benign control values are percolated forward at each clock cycle with the proper effect: no registers or memories are written if the control values are all 0.

The figure below shows what really happens in the hardware: the pipeline execution slot associated with the `AND` instruction is turned into a nop and all instructions beginning with the `AND` instruction are delayed one cycle. Like an air bubble in a water pipe, a stall bubble delays everything behind it and proceeds down the instruction pipe one stage each clock cycle until it exits at the end. In this example, the hazard forces the `AND` and `ORR` instructions to repeat in clock cycle 4 what they did in clock cycle 3: `AND` reads registers and decodes, and `ORR` is refetched from instruction memory. Such repeated work is what a stall looks like, but its effect is to stretch the time of the `AND` and `ORR` instructions and delay the fetch of the `ADD` instruction.

Figure 4.7.6: The way stalls are really inserted into the pipeline (COD Figure 4.58).

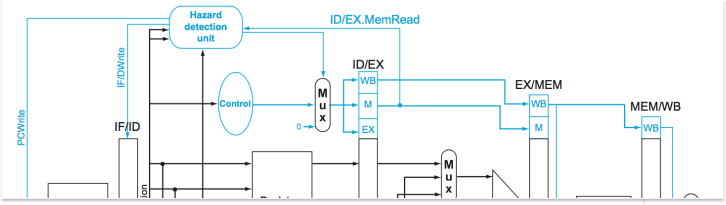
A bubble is inserted beginning in clock cycle 4, by changing the `AND` instruction to a nop. Note that the `AND` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise, the `ORR` instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.



The figure below highlights the pipeline connections for both the hazard detection unit and the forwarding unit. As before, the forwarding unit controls the ALU multiplexors to replace the value from a general-purpose register with the value from the proper pipeline register. The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0s. The hazard detection unit stalls and deasserts the control fields if the load-use hazard test above is true. If you would like to see more details, COD Section 4.13 (Advanced topic: An introduction to digital design using a hardware design language …) gives an example illustrated using single-clock pipeline diagrams of LEGv8 code with hazards that cause stalling.

Figure 4.7.7: Pipelined control overview, showing the two multiplexor for forwarding, the hazard detection unit, and the forwarding unit (COD Figure 4.59).

Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.



The Big Picture

Although the compiler generally relies upon the hardware to resolve hazards and thereby ensure correct execution, the compiler must understand the pipeline to achieve the best performance. Otherwise, unexpected stalls will reduce the performance of the compiled code.

Elaboration

Regarding the remark earlier about setting control lines to 0 to avoid writing registers or memory: only the signals RegWrite and MemWrite need be 0, while the other control signals can be don't cares.

PARTICIPATION ACTIVITY 4.7.8: Hazard detection unit.

Match the hazard detection conditions and actions to the corresponding code snippet.

Detects a load instruction.

The register written by the load is read by the next instruction's second operand.

Inserts a bubble.

The register written by the load is read by the next instruction's first operand.

---

if (ID/EX.MemRead and

((ID/EX.RegisterRm2 = IF/ID.RegisterRn1) or

(ID/EX.RegisterRm2 = IF/ID.RegisterRm2)))

stall the pipeline

Reset

Provide feedback on this section