


With remarkable advances in processing, Amdahl's law suggests that another part of the system will become the bottleneck. That bottleneck is the topic of the next chapter: the memory system.

An alternative to pushing uniprocessors to automatically exploit parallelism at the instruction level is trying multiprocessors, which exploit parallelism at much coarser levels. Parallel processing is the topic of  Chapter 9, which appears on the CD.

## 6.13

### Historical Perspective and Further Reading

This section, which appears on the CD, discusses the history of the first pipelined processors, the earliest superscalars, the development of out-of-order and speculative techniques, as well as important developments in the accompanying compiler technology.

## 6.14

### Exercises

**6.1** [5] <§6.1> If the time for an ALU operation can be shortened by 25% (compared to the description in Figure 6.2 on page 373);

- Will it affect the speedup obtained from pipelining? If yes, by how much? Otherwise, why?
- What if the ALU operation now takes 25% more time?

**6.2** [10] <§6.1> A computer architect needs to design the pipeline of a new microprocessor. She has an example workload program core with  $10^6$  instructions. Each instruction takes 100 ps to finish.

- How long does it take to execute this program core on a nonpipelined processor?
- The current state-of-the-art microprocessor has about 20 pipeline stages. Assume it is perfectly pipelined. How much speedup will it achieve compared to the nonpipelined processor?
- Real pipelining isn't perfect, since implementing pipelining introduces some overhead per pipeline stage. Will this overhead affect instruction latency, instruction throughput, or both?

**6.3** [5] <§6.1> Using a drawing similar to Figure 6.5 on page 377, show the forwarding paths needed to execute the following four instructions:

```
add $3, $4, $6
sub $5, $3, $2
lw  $7, 100($5)
add $8, $7, $2
```

**6.4** [10] <§6.1> Identify all of the data dependencies in the following code. Which dependencies are data hazards that will be resolved via forwarding? Which dependencies are data hazards that will cause a stall?


```
add $3, $4, $2
sub $5, $3, $1
lw  $6, 200($3)
add $7, $3, $6
```

**6.5** [5] <§6.1>  For More Practice: Delayed Branches

**6.6** [10] <§6.2> Using Figure 6.22 on page 400 as a guide, use colored pens or markers to show which portions of the datapath are active and which are inactive in each of the five stages of the `sw` instruction. We suggest that you use five photocopies of Figure 6.22 to answer this exercise. (We hereby grant you permission to violate the Copyright Protection Act in doing the exercises in Chapters 5 and 6!) Be sure to include a legend to explain your color scheme.

**6.7** [5] <§6.2>  For More Practice: Understanding Pipelines by Drawing Them


**6.8** [5] <§6.2>  For More Practice: Understanding Pipelines by Drawing Them

**6.9** [15] <§6.2>  For More Practice: Understanding Pipelines by Drawing Them

**6.10** [5] <§6.2>  For More Practice: Pipeline Registers

**6.11** [15] <§§4.8, 6.2>  For More Practice: Pipelining Floating Point

**6.12** [15] <§6.3> Figure 6.37 on page 417 and Figure 6.35 on page 415 are two styles of drawing pipelines. To make sure you understand the relationship between these two styles, draw the information in Figures 6.31 through 6.35 on pages 410 through 415 using the style of Figure 6.37 on page 417. Highlight the active portions of the data paths in the figure.

**6.13** [20] <§6.3> Figure 6.14.10 is similar to Figure 6.14.7 on page 6.14-9 in the  **For More Practice** section, but the instructions are unidentified. Determine as much as you can about the five instructions in the five pipeline stages. If you cannot fill in a field of an instruction, state why. For some fields it will be easier to decode the machine instructions into assembly language, using Figure 3.18 on

page 205 and Figure A.10.2 on page A-50 as references. For other fields it will be easier to look at the values of the control signals, using Figures 6.26 through 6.28 on pages 403 and 405 as references. You may need to carefully examine Figures 6.14.5 through 6.14.9 to understand how collections of control values are presented (i.e., the leftmost bit in one cycle will become the uppermost bit in another cycle). For example, the EX control value for the subtract instruction, 1100, computed during the ID stage of cycle 3 in Figure 6.14.6, becomes three separate values specifying RegDst (1), ALUOp (10), and ALUSrc (0) in cycle 4.

**6.14** [40] <§6.3> The following piece of code is executed using the pipeline shown in Figure 6.30 on page 409:

```
lw $5, 40($2)
add $6, $3, $2
or $7, $2, $1
and $8, $4, $3
sub $9, $2, $1
```

At cycle 5, right before the instructions are executed, the processor state is as follows:

- The PC has the value  $100_{\text{ten}}$ , the address of the `sub` instruction.
- Every register has the initial value  $10_{\text{ten}}$  plus the register number (e.g., register \$8 has the initial value  $18_{\text{ten}}$ ).
- Every memory word accessed as data has the initial value  $1000_{\text{ten}}$  plus the byte address of the word (e.g., `Memory[8]` has the initial value  $1008_{\text{ten}}$ ).

Determine the value of every field in the four pipeline registers in cycle 5.

**6.15** [20] <§6.3>  For More Practice: Labeling Pipeline Diagrams with Control

**6.16** [20] <§6.4>  For More Practice: Illustrating Diagrams with Forwarding

**6.17** [5] <§§6.4, 6.5> Consider executing the following code on the pipelined datapath of Figure 6.36 on page 416:

```
add $2, $3, $1
sub $4, $3, $5
add $5, $3, $7
add $7, $6, $1
add $8, $2, $6
```

At the end of the fifth cycle of execution, which registers are being read and which register will be written?

**6.18** [5] <§§6.4, 6.5> With regard to the program in Exercise 6.17, explain what the forwarding unit is doing during the fifth cycle of execution. If any comparisons are being made, mention them.

**6.19** [5] <§§6.4, 6.5> With regard to the program in Exercise 6.17, explain what the hazard detection unit is doing during the fifth cycle of execution. If any comparisons are being made, mention them.

**6.20** [20] <§§6.4, 6.5>  For More Practice: Forwarding in Memory

**6.21** [5] <§6.5> We have a program of  $10^3$  instructions in the format of "lw, add, lw, add, ..." The add instruction depends (and only depends) on the lw instruction right before it. The lw instruction also depends (and only depends) on the add instruction right before it. If the program is executed on the pipelined datapath of Figure 6.36 on page 416:

- What would be the actual CPI?
- Without forwarding, what would be the actual CPI?


**6.22** [5] <§§6.4, 6.5> Consider executing the following code on the pipelined datapath of Figure 6.36 on page 416:

```
lw    $4, 100($2)
sub   $6, $4, $3
add   $2, $3, $5
```

How many cycles will it take to execute this code? Draw a diagram like that of Figure 6.34 on page 414 that illustrates the dependencies that need to be resolved, and provide another diagram like that of Figure 6.35 on page 415 that illustrates how the code will actually be executed (incorporating any stalls or forwarding) so as to resolve the identified problems.

**6.23** [15] <§6.5> List all the inputs and outputs of the forwarding unit in Figure 6.36 on page 416. Give the names, the number of bits, and brief usage for each input and output.

**6.24** [20] <§6.5>  For More Practice: Illustrating Diagrams with Forwarding and Stalls

**6.25** [20] <§6.5>  For More Practice: Impact on Forwarding of Moving It to ID Stage

**6.26** [15] <§§6.2–6.5>  For More Practice: Impact of Memory Addressing Mode on Pipeline

**6.27** [10] <§§6.2–6.5>  For More Practice: Impact of Arithmetic Operations with Memory Operands on Pipeline


**6.28** [30] <§6.5, Appendix C>  For More Practice: Forwarding Unit Hardware Design


**6.29** [1 week] <§§6.4, 6.5> Using the simulator provided with this book, collect statistics on data hazards for a C program (supplied by either the instructor or with the software). You will write a subroutine that is passed the instruction to be executed, and this routine must model the five-stage pipeline in this chapter. Have your program collect the following statistics:

- Number of instructions executed.
- Number of data hazards not resolved by forwarding and number resolved by forwarding.
- If the MIPS C compiler that you are using issues `nop` instructions to avoid hazards, count the number of `nop` instructions as well.

Assuming that the memory accesses always take 1 clock cycle, calculate the average number of clock cycles per instruction. Classify `nop` instructions as stalls inserted by software, then subtract them from the number of instructions executed in the CPI calculation.

**6.30** [7] <§§6.4, 6.5> In the example on page 425, we saw that the performance advantage of the multicycle design was limited by the longer time required to access memory versus use the ALU. Suppose the memory access became 2 clock cycles long. Find the relative performance of the single-cycle and multicycle designs. In the next few exercises, we extend this to the pipelined design, which requires lots more work!

**6.31** [10] <§6.6>  For More Practice: Coding with Conditional Moves

**6.32** [10] <§6.6>  For More Practice: Performance Advantage of Conditional Move

**6.33** [20] <§§6.2–6.6> In the example on page 425, we saw that the performance advantage of both the multicycle and the pipelined designs was limited by the longer time required to access memory versus use the ALU. Suppose the memory access became 2 clock cycles long. Draw the modified pipeline. List all the possible new forwarding situations and all possible new hazards and their length.

**6.34** [20] <§§6.2–6.6> Redo the example on page 425 using the restructured pipeline of Exercise 6.33 to compare the single-cycle and multicycle. For branches, assume the same prediction accuracy, but increase the penalty as appropriate. For loads, assume that the subsequent instructions depend on the load with a probability of 1/2, 1/4, 1/8, 1/16, and so on. That is, the instruction following a load by two has a 25% probability of using the load result as one of its sources. Ignoring any other data hazards, find the relative performance of the pipelined design to the single-cycle design with the restructured pipeline.

**6.35** [10] <§§6.4–6.6> As pointed out on page 418, moving the branch comparison up to the ID stage introduces an opportunity for both forwarding and hazards that cannot be resolved by forwarding. Give a set of code sequences that show the possible

forwarding paths required and hazard cases that must be detected, considering only one of the two operands. The number of cases should equal the maximum length of the hazard if no forwarding existed.

**6.36** [15] <§6.6> We have a program core consisting of five conditional branches. The program core will be executed thousands of times. Below are the outcomes of each branch for one execution of the program core (T for taken, N for not taken).

Branch 1: T-T-T  
 Branch 2: N-N-N-N  
 Branch 3: T-N-T-N-T-N  
 Branch 4: T-T-T-N-T  
 Branch 5: T-T-N-T-T-N-T

Assume the behavior of each branch remains the same for each program core execution. For dynamic schemes, assume each branch has its own prediction buffer and each buffer initialized to the same state before each execution. List the predictions for the following branch prediction schemes:

- Always taken
- Always not taken
- 1-bit predictor, initialized to predict taken
- 2-bit predictor, initialized to weakly predict taken

What are the prediction accuracies?

**6.37** [10] <§§6.4–6.6> Sketch all the forwarding paths for the branch inputs and show when they must be enabled (as we did on page 407).

**6.38** [10] <§§6.4–6.6> Write the logic to detect any hazards on the branch sources, as we did on page 410.

**6.39** [10] <§§6.4–6.6> The example on page 378 shows how to *maximize* performance on our pipelined datapath with forwarding and stalls on a use following a load. Rewrite the following code to *minimize* performance on this datapath—that is, reorder the instructions so that this sequence takes the *most* clock cycles to execute while still obtaining the same result.

```
lw    $2, 100($6)
lw    $3, 200($7)
add   $4, $2, $3
add   $6, $3, $5
sub   $8, $4, $6
lw    $7, 300($8)
beq   $7, $8, Loop
```

**6.40** [20] <§6.6> Consider the pipelined datapath in Figure 6.54 on page 461. Can an attempt to flush and an attempt to stall occur simultaneously? If so, do they result in conflicting actions and/or cooperating actions? If there are any cooperating actions, how do they work together? If there are any conflicting actions, which should take priority? Is there a simple change you can make to the datapath to ensure the necessary priority? You may want to consider the following code sequence to help you answer this question:

```

        beq $1, $2, TARGET    # assume that the branch is taken
        lw  $3, 40($4)
        add $2, $3, $4
        sw  $2, 40($4)
TARGET: or  $1, $1, $2

```

**6.41** [15] <§§6.4, 6.7> The Verilog for implementing forwarding in Figure 6.7.2 on page 6.7-4–6.7-5 did not consider forwarding of a result as the value to be stored by a SW instruction. Add this to the Verilog code.

**6.42** [5] <§§6.5, 6.7> The Verilog for implementing stalls in Figure 6.7.3 on page 6.7-6–6.7-7 did not consider forwarding of a result to use in an address calculation. Make this simple addition to the Verilog code.

**6.43** [15] <§§6.6, 6.7> The Verilog code for implementing branch hazard detection and stalls in Figure 6.7.3 on page 6.7-6–6.7-7 does not detect the possibility of data hazards for the two source registers of a BEQ instruction. Extend the Verilog in Figure 6.7.3 on page 6.7-6–6.7-7 to handle all data hazards for branch operands. Write both the forwarding and stall logic needed for completing branches during ID.

**6.44** [10] <§§6.6, 6.7> Rewrite the Verilog code in 6.7.3 on page 6.7-6–6.7-7 to implement a delayed branch strategy.

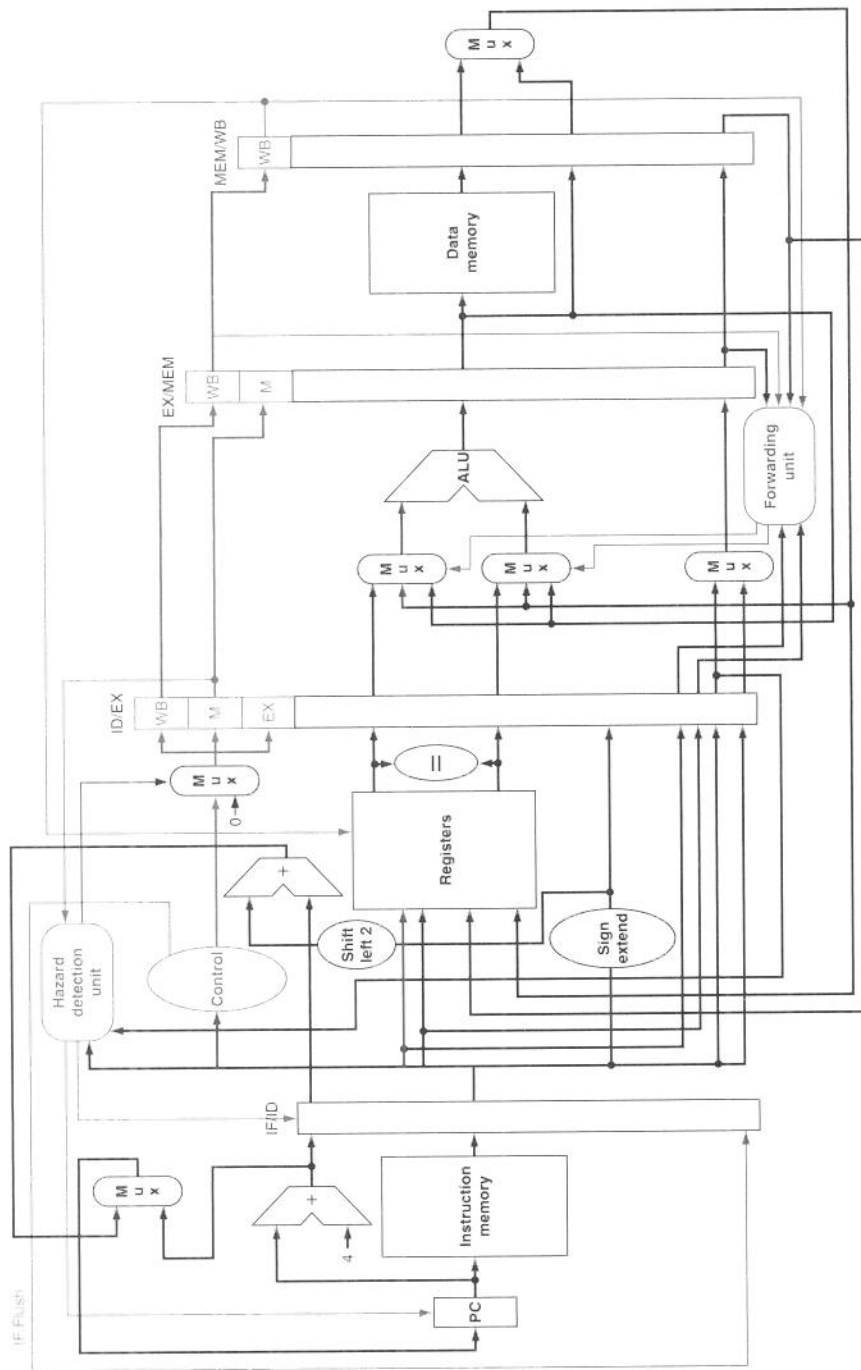
**6.45** [20] <§§6.6, 6.7> Rewrite the verilog code in Figure 6.7.3 on page 6.7-6–6.7-7 to implement a branch target buffer. Assume the buffer is implemented with a module with the following definition:

```

module PredictPC (currentPC,nextPC,miss,update,destination):
    input currentPC,
        update, // true if previous prediction was unavailable or incorrect
        destination; // used with update to correct a prediction
    output nextPC, // returns the next PC if prediction is accurate
        miss; // true means no prediction in buffer
endmodule;

```

Make sure you accommodate all three possibilities: a correct prediction, a miss in the buffer (that is, miss = true), and an incorrect prediction. In the last two cases, you must also update the prediction.



**FIGURE 6.54 Datapath for branch, including hardware to flush the instruction that follows the branch.** This optimization moves the branch decision from the fourth pipeline stage to the second; only one instruction that follows the branch will be in the pipe at that time. The control line IF-Flush turns the fetched instruction into a nop by zeroing the IF/ID pipeline register. Although the flush line is shown coming from the control unit in this figure, in reality it comes from hardware that determines if a branch is taken, labeled with an equal sign to the right of the registers in the ID stage. The forwarding muxes and paths must also be added to this stage, but are not shown to simplify the figure.



**6.46** [1 month] <§5.4, 6.3–6.8> If you have access to a simulation system such as Verilog or ViewLogic, first design the single-cycle datapath and control from Chapter 5. Then evolve this design into a pipelined organization, as we did in this chapter. Be sure to run MIPS programs at each step to ensure that your refined design continues to operate correctly.

**6.47** [10] <§6.9> The following code has been unrolled once but not yet scheduled. Assume the loop index is a multiple of two (i.e., \$10 is a multiple of eight):

```
Loop:   lw    $2, 0($10)
        sub  $4, $2, $3
        sw  $4, 0($10)
        lw  $5, 4($10)
        sub  $6, $5, $3
        sw  $6, 4($10)
        addi $10, $10, 8
        bne $10, $30, Loop
```



Schedule this code for fast execution on the standard MIPS pipeline (assume that it supports `addi` instruction). Assume initially \$10 is 0 and \$30 is 400 and that branches are resolved in the MEM stage. How does the scheduled code compare against the original unscheduled code?

**6.48** [20] <§6.9> This exercise is similar to Exercise 6.47, except this time the code should be unrolled twice (creating three copies of the code). However, it is not known that the loop index is a multiple of three, and thus you will need to invent a means of ensuring that the code still executes properly. (Hint: Consider adding some code to the beginning or end of the loop that takes care of the cases not handled by the loop.)

**6.49** [20] <§6.9> Using the code in Exercise 6.47, unroll the code four times and schedule it for the static multiple-issue version of the MIPS processor described on pages 436–439. You may assume that the loop executes for a multiple of four times.

**6.50** [10] <§6.1–6.9> As technology leads to smaller feature sizes, the wires become relatively slower (as compared to the logic). As logic becomes faster with the shrinking feature size and clock rates increase, wire delays consume more clock cycles. That is why the Pentium 4 has several pipeline stages dedicated to transferring data along wires from one part of the pipeline to another. What are the drawbacks to having to add pipe stages for wire delays?

**6.51** [30] <§6.10> New processors are introduced more quickly than new versions of textbooks. To keep your textbook current, investigate some of the latest developments in this area and write a one-page elaboration to insert at the end of Section 6.10. Use the World-Wide Web to explore the characteristics of the latest processors from Intel or AMD as a starting point.

- §6.1, page 384: 1. Stall on the LW result. 2. Bypass the ADD result. 3. No stall or bypass required.
- §6.2, page 399: Statements 2 and 5 are correct; the rest are incorrect.
- §6.6, page 426: 1. Predict not taken. 2. Predict taken. 3. Dynamic prediction.
- §6.7,  page 6.7-3: Statements 1 and 3 are both true.
- §6.7,  page 6.7-7: Only statement #3 is completely accurate.
- §6.8, page 432: Only #4 is totally accurate. #2 is partially accurate.
- §6.9, page 447: Speculation: both; reorder buffer: hardware; register renaming: both; out-of-order execution: hardware; predication: software; branch prediction: both; VLIW: software; superscalar: hardware; EPIC: both, since there is substantial hardware support; multiple issue: both; dynamic scheduling: hardware.
- §6.10, page 450: All the statements are false.

**Answers to  
Check Yourself**