Chapter 2 Case Studies with Alternate Exercises by Robert P. Colwell

Case Study 1: Exploring the Impact of Microarchitectural Techniques

Concepts illustrated by this case study

- Basic Instruction Scheduling, Reordering, Dispatch
- Multiple Issue and Hazards
- Register Renaming
- Out-of-Order and Speculative Execution
- Where to Spend Out-of-Order Resources

You are tasked with designing a new processor microarchitecture, and you are trying to figure out how best to allocate your hardware resources. Which of the hardware and software techniques you learned in Chapter 2 should you apply? You have a list of latencies for the functional units and for memory, as well as some representative code. Your boss has been somewhat vague about the performance requirements of your new design, but you know from experience that, all else being equal, faster is usually better. Start with the basics. Figure 2.35 provides a sequence of instructions and list of latencies.

```
Loop: LD     F2,0(Rx)
I0:   DIVD   F8,F2,F0
I1:   MULTD  F2,F6,F2
I2:   LD     F4,0(Ry)
I3:   ADDD   F4,F0,F4
I4:   ADDD   F10,F8,F2
I5:   ADDI   Rx,Rx,#8
I6:   ADDI   Ry,Ry,#8
I7:   SD     F4,0(Ry)
I8:   SUB    R20,R4,Rx
I9:   BNZ    R20,Loop
```

Latencies beyond single cycle	
Memory LD	+4
Memory SD	+1
Integer ADD, SUB	+0
Branches	+1
ADDD	+1
MULTD	+5
DIVD	+12

Figure 2.35 Code and latencies for Exercises 2.1 through 2.6.

2.1 [10] <1.8, 2.1, 2.2> What would be the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 2.35, if no new instruction's execution could be initiated until the previous instruction's execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one cycle branch delay slot.

2.2 [10] <1.8, 2.1, 2.2> Think about what latency numbers really mean—they indicate the number of cycles a given function requires to produce its output, nothing more. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a "producer" followed by a "consumer") will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 2.35 require if the pipeline detected true data dependencies and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? Show the code with `<stall>` inserted where necessary to accommodate stated latencies. (*Hint:* An instruction with latency "+2" needs 2 `<stall>` cycles to be inserted into the code sequence. Think of it this way: a 1-cycle instruction has latency 1 + 0, meaning zero extra wait states. So latency 1 + 1 implies 1 stall cycle; latency 1 + *N* has *N* extra stall cycles.)

2.3 [15] <2.6, 2.7> Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require?

2.4 [10] <2.6, 2.7> In the multiple-issue design of Exercise 2.3, you may have recognized some subtle issues. Even though the two pipelines have the exact same instruction repertoire, they are not identical nor interchangeable, because there is an implicit ordering between them that must reflect the ordering of the instructions in the original program. If instruction *N* + 1 begins execution in Execution Pipe 1 at the same time that instruction *N* begins in Pipe 0, and *N* + 1 happens to require a shorter execution latency than *N,* then *N* + 1 will complete before *N* (even though program ordering would have implied otherwise). Recite at least two reasons why that could be hazardous and will require special considerations in the microarchitecture. Give an example of two instructions from the code in Figure 2.35 that demonstrate this hazard.

2.5 [20] <2.7> Reorder the instructions to improve performance of the code in Figure 2.35. Assume the two-pipe machine in Exercise 2.3, and that the out-of-order completion issues of Exercise 2.4 have been dealt with successfully. Just worry about observing true data dependencies and functional unit latencies for now. How many cycles does your reordered code take?

2.6 [10/10] <2.1, 2.2> Every cycle that does not initiate a new operation in a pipe is a lost opportunity, in the sense that your hardware is not "living up to its potential."

a. [10] <2.1, 2.2> In your reordered code from Exercise 2.5, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?

b. [10] <2.1, 2.2> Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance. Hand-unroll two iterations of the loop in your reordered code from Exercise 2.5.

c. [10] <2.1, 2.2> What speedup did you obtain? (For this exercise, just color the $N + 1$ iteration's instructions green to distinguish them from the Nth iteration's; if you were actually unrolling the loop you would have to reassign registers to prevent collisions between the iterations.)

2.7 [15] <2.1> Computers spend most of their time in loops, so multiple loop iterations are great places to speculatively find more work to keep CPU resources busy. Nothing is ever easy, though; the compiler emitted only one copy of that loop's code, so even though multiple iterations are handling distinct data, they will appear to use the same registers. To keep multiple iterations' register usages from colliding, we rename their registers. Figure 2.36 shows example code that we would like our hardware to rename.

A compiler could have simply unrolled the loop and used different registers to avoid conflicts, but if we expect our hardware to unroll the loop, it must also do the register renaming. How? Assume your hardware has a pool of temporary registers (call them T registers, and assume there are 64 of them, T0 through T63) that it can substitute for those registers designated by the compiler. This rename hardware is indexed by the src (source) register designation, and the value in the table is the T register of the last destination that targeted that register. (Think of these table values as producers, and the src registers are the consumers; it doesn't much matter where the producer puts its result as long as its consumers can find it.) Consider the code sequence in Figure 2.36. Every time you see a destination register in the code, substitute the next available T, beginning with T9. Then update all the src registers accordingly, so that true data dependencies are maintained. Show the resulting code. (*Hint:* See Figure 2.37.)

```
Loop: LD     F4,0(Rx)
I0:   MULTD  F2,F0,F2
I1:   DIVD   F8,F4,F2
I2:   LD     F4,0(Ry)
I3:   ADDD   F6,F0,F4
I4:   SUBD   F8,F8,F6
I5:   SD     F8,0(Ry)
```

Figure 2.36 Sample code for register renaming practice.

```
I0:  LD     T9,0(Rx)
I1:  MULTD  T10,F0,T9
. . .
```

Figure 2.37 Hint: expected output of register renaming.

```
I0:  SUBD   F1,F2,F3
I1:  ADDD   F4,F1,F2
I2:  MULTD  F6,F4,F1
I3:  DIVD   F0,F2,F6
```

Figure 2.38 Sample code for superscalar register renaming.

2.8 [20] <2.4> Exercise 2.7 explored simple register renaming: when the hardware register renamer sees a source register, it substitutes the destination T register of the last instruction to have targeted that source register. When the rename table sees a destination register, it substitutes the next available T for it. But superscalar designs need to handle multiple instructions per clock cycle at every stage in the machine, including the register renaming. A simple scalar processor would therefore look up both src register mappings for each instruction, and allocate a new dest mapping per clock cycle. Superscalar processors must be able to do that as well, but they must also ensure that any dest-to-src relationships between the two concurrent instructions are handled correctly. Consider the sample code sequence in Figure 2.38. Assume that we would like to simultaneously rename the first two instructions. Further assume that the next two available T registers to be used are known at the beginning of the clock cycle in which these two instructions are being renamed. Conceptually, what we want is for the first instruction to do its rename table lookups, and then update the table per its destination's T register. Then the second instruction would do exactly the same thing, and any interinstruction dependency would thereby be handled correctly. But there's not enough time to write that T register designation into the renaming table and then look it up again for the second instruction, all in the same clock cycle. That register substitution must instead be done live (in parallel with the register rename table update). Figure 2.39 shows a circuit diagram, using multiplexers and comparators, that will accomplish the necessary on-the-fly register renaming. Your task is to show the cycle-by-cycle state of the rename table for every instruction of the code shown in Figure 2.38. Assume the table starts out with every entry equal to its index (T0 = 0; T1 = 1, . . .).

2.9 [10] <2.4> If you ever get confused about what a register renamer has to do, go back to the assembly code you're executing, and ask yourself what has to happen for the right result to be obtained. For example, consider a three-way superscalar machine renaming these three instructions concurrently:

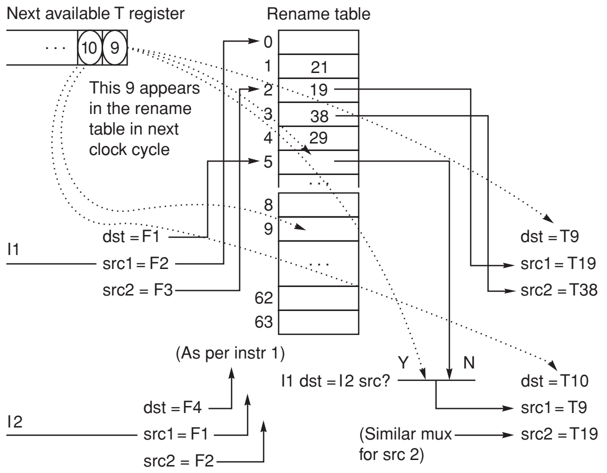

Figure 2.39 Rename table and on-the-fly register substitution logic for superscalar machines.

```
ADDI    R1, R1, R1
ADDI    R1, R1, R1
ADDI    R1, R1, R1
```

If the value of R1 starts out as 5, what should its value be when this sequence has executed?

2.10 [20] <2.4, 2.9> VLIW designers have a few basic choices to make regarding architectural rules for register use. Suppose a VLIW is designed with self-draining execution pipelines: once an operation is initiated, its results will appear in the destination register at most *L* cycles later (where *L* is the latency of the operation). There are never enough registers, so there is a temptation to wring maximum use out of the registers that exist. Consider Figure 2.40. If loads have a 1 + 2 cycle latency, unroll this loop once, and show how a VLIW capable of two loads and two adds per cycle can use the minimum number of registers, in the absence of any pipeline interruptions or stalls. Give an example of an event that, in the presence of self-draining pipelines, could disrupt this pipelining and yield wrong results.

2.11 [10/10/10] <2.3> Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write back) and the code in Figure 2.41. All ops are 1 cycle except `LW` and `SW`, which are 1 + 2 cycles, and branches, which are 1 + 1 cycles. There is no forwarding. Show the phases of each instruction per clock cycle for one iteration of the loop.

a. [10] <2.3> How many clock cycles per loop iteration are lost to branch overhead?

```
Loop: LW     R4,0(R0) ;   ADDI    R11,R3,#1
      LW     R5,8(R1) ;   ADDI    R20,R0,#1
      <stall>
      ADDI   R10,R4,#1;
      SW     R7,0(R6) ;   SW      R9,8(R8)
      ADDI   R2,R2,#8
      SUB    R4,R3,R2
      BNZ    R4,Loop
```

Figure 2.40 Sample VLIW code with two adds, two loads, and two stalls.

```
Loop: LW     R3,0(R0)
      LW     R1,0(R3)
      ADDI   R1,R1,#1
      SUB    R4,R3,R2
      SW     R1,0(R3)
      BNZ    R4, Loop
```

Figure 2.41 Code loop for Exercise 2.11.

b. [10] <2.3> Assume a static branch predictor, capable of recognizing a backwards branch in the Decode stage. Now how many clock cycles are wasted on branch overhead?

c. [10] <2.3> Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?

2.12 Let's consider what dynamic scheduling might achieve here. Assume a microarchitecture as shown in Figure 2.42. Assume that the ALUs can do all arithmetic ops (`MULTD`, `DIVD`, `ADDD`, `ADDI`, `SUB`) and branches, and that the RS can dispatch at most one operation to each functional unit per cycle (one op to each ALU plus one memory op to the LD/ST unit).

a. [15] <2.4> Suppose all of the instructions from the sequence in Figure 2.35 are present in the RS, with no renaming having been done. Highlight any instructions in the code where register renaming would improve performance. *Hint:* Look for RAW and WAW hazards. Assume the same functional unit latencies as in Figure 2.35.

b. [20] <2.4> Suppose the register-renamed version of the code from part (a) is resident in the RS in clock cycle *N*, with latencies as given in Figure 2.35. Show how the RS should dispatch these instructions out-of-order, clock by clock, to obtain optimal performance on this code. (Assume the same RS restrictions as in part (a). Also assume that results must be written into the RS

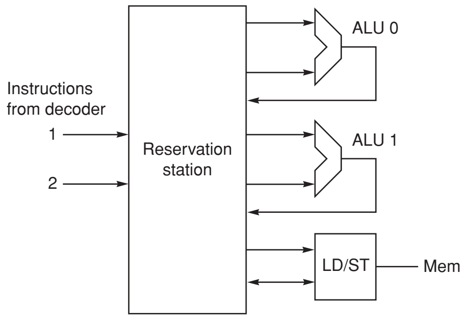

Figure 2.42 An out-of-order microarchitecture.

before they're available for use; i.e., no bypassing.) How many clock cycles does the code sequence take?

c. [20] <2.4> Part (b) lets the RS try to optimally schedule these instructions. But in reality, the whole instruction sequence of interest is not usually present in the RS. Instead, various events clear the RS, and as a new code sequence streams in from the decoder, the RS must choose to dispatch what it has. Suppose that the RS is empty. In cycle 0 the first two register-renamed instructions of this sequence appear in the RS. Assume it takes 1 clock cycle to dispatch any op, and assume functional unit latencies are as they were for Exercise 2.2. Further assume that the front end (decoder/register-renamer) will continue to supply two new instructions per clock cycle. Show the cycle-by-cycle order of dispatch of the RS. How many clock cycles does this code sequence require now?

d. [10] <2.10> If you wanted to improve the results of part (c), which would have helped most: (1) another ALU; (2) another LD/ST unit; (3) full bypassing of ALU results to subsequent operations; (4) cutting the longest latency in half? What's the speedup?

e. [20] <2.7> Now let's consider speculation, the act of fetching, decoding, and executing beyond one or more conditional branches. Our motivation to do this is twofold: the dispatch schedule we came up with in part (c) had lots of nops, and we know computers spend most of their time executing loops (which implies the branch back to the top of the loop is pretty predictable.) Loops tell us where to find more work to do; our sparse dispatch schedule suggests we have opportunities to do some of that work earlier than before. In part (d) you found the critical path through the loop. Imagine folding a second copy of that path onto the schedule you got in part (b). How many more clock cycles would be required to do two loops' worth of work (assuming all instructions are resident in the RS)? (Assume all functional units are fully pipelined.)