ECE 4750 Computer Architecture

Topic 11: Advanced Processors – Speculative Execution

http://www.csl.cornell.edu/courses/ece4750
School of Electrical and Computer Engineering
Cornell University

revision: 2022-11-30-12-41

List of Problems

Problem 1. Short Answer

Part 1.A Speculative Execution in IO2E Microarchitecture

Consider the canonical single-issue IO2E microarchitecture with an in-order front-end and out-of-order issue/writeback and early commit (see Figure A.4 in Appendix A). Recall that this basic microarchitecture does not have an ROB, register renaming, nor memory disambiguation. Assume we wish to add support for executing a single branch in IO2E with speculative execution: branches are resolved in the X stage, and we add speculative bits so that speculative instructions after the branch can issue/execute/writeback but can also be squashed on a mispredicted branch. Since there is no register renaming we cannot snapshot the rename table. Unfortunately, adding support for executing a branch in IO2E with speculative execution can lead to an incorrect value being used in a non-speculative arithmetic instruction.

Create a sequence of assembly instructions and corresponding pipeline diagram which clearly illustrates what can go wrong. Briefly explain the problem. *You do not need to craft dependencies to force instructions to stall or issue on a specific cycle*; simply have instructions fetch, decode, and then arbitrarily wait in the issue queue until you want them to issue. *Please note, that since we are assuming no register renaming your instruction sequence should not include any WAW or WAR dependencies!* Your example should be as simple as possible to illustrate the problem. Your assembly sequence obviously needs to include a branch; *assume that we incorrectly predict this branch as not-taken, and thus we must redirect control flow in the X stage to the branch target and squash all speculative instructions.* **You should include the execution of the branch target in your pipeline diagram. Draw a clearly labeled arrow illustrating the problem: the arrow should end at the stage that reads the incorrect value and the arrow should start at the stage where that value was written.**

Explain what modifications need to be made to the IO2L microarchitecture to enable the pipeline diagram on the previous page and thus enable correct execution of `movz` instructions. If possible, your modifications should fit within the mechanisms already provided in the complete quad-issue IO2L microarchitecture shown in Figure A.7 (e.g., your modifications should not require any new data structures). If you need to modify a data structure, feel free to sketch the modified data structure below to more clearly indicate what changes need to be made.

Part 1.B Exceptions in an IO2L Microarchitecture

Consider the complete *single-issue* IO2L microarchitecture with an in-order front-end and out-of-order issue/writeback with late commit (see Figure A.7 in Appendix A). This microarchitecture includes pointer-based register renaming, memory disambiguation with out-of-order load/store issue, branch prediction, and speculative execution.

Assume that we modify the semantics of the TinyRV2 integer multiply instruction. The new semantics are such that when the result of an integer multiply overflows (i.e., the result is larger than what can be held in a 32-bit register) it causes a hardware exception. The overflow condition is detected in stage Y3.

Draw a pipeline diagram that illustrates the execution of the assembly code sequence below *if instruction 4 experiences a multiplication overflow exception in stage Y3.* Clearly indicate which instructions are killed and when they are killed by using a forward slash symbol (/) on the cycle an instruction is killed. Your pipeline diagram should end with the decode of `opA`. **Draw a control dependency arrow to indicate the control flow. Clearly label what stage exceptions are handled in.**

```
1   mul  x1,  x2, x3
2   mul  x4,  x1, x5
3   addi x9,  x10, 1
4   mul  x6,  x7, x8   # experiences a multiplication overflow exception
5   addi x11, x12, 1
6   addi x13, x14, 1
7   addi x15, x16, 1
8   addi x17, x18, 1
9   ...
10 exception_handler:
11  opA
```

`mul  x1, x2, x3`																		
`mul  x4, x1, x5`																		
`addi x9, x10, 1`																		
`mul  x6, x7, x8`																		
`addi x11, x12, 1`																		
`addi x13, x14, 1`																		
`addi x15, x16, 1`																		
`addi x17, x18, 1`																		
`...`																		
`opA`																		

Part 1.C Out-of-Order Superscalar Processors

We wish to execute the following short assembly code sequence processing 64 elements. It would probably be useful to make sure you thoroughly understand this code and the architectural dependencies before continuing. Note that we have rescheduled the address base pointer increment and the loop counter decrement before the store.

```
1         loop:
2   0x100  lw   x1, 0(x2)
3   0x104  lw   x3, 0(x1)    # address depends on value loaded above
4   0x108  mul  x4, x3, x5
5   0x10c  addi x2, x2, 4    # ptr increment scheduled here to optimize performance
6   0x110  addi x6, x6, -1   # assume x6 initially is 64
7   0x114  sw   x4, -4(x2)   # negative offset because ptr increment is above
8   0x118  bne  x6, x0, loop
```

Consider the canonical *quad-issue* IO2L microarchitecture with an in-order front-end and out-of-order issue/writeback with late commit (see Figure A.7 in Appendix A). Note that there are four functional units (Y-pipe for multiplies, X-pipe for short-latency integer ops, L-pipe for loads, and S-pipe for stores); this microarchitecture only provides a single short-latency integer ALU (i.e., the X-pipe). This microarchitecture includes support for register renaming and an aggressive memory disambiguation scheme that enables loads and stores to issue out-of-order. Assume we use unified stores such that both the store data and store address must be ready before we can issue a store. Assume that we have an infinite number of entries in the various data structures (e.g., issue queue, physical register file, reorder buffer, finished store buffer, etc). *Assume that there are no instruction nor data cache misses and assume perfect branch prediction (i.e., dynamic branch predictors always correctly predict the right control flow path in the fetch stage resulting in no branch resolution penalty).* Do not assume that all of the instructions are waiting in the issue queue. You must explicitly fetch and decode instructions in-order.

Draw a pipeline diagram illustrating the execution of two iterations of the given assembly loop on this microarchitecture. Estimate the total execution time of the entire loop on this microarchitecture. The instructions for the first iteration are already filled in for you. Normally, we would need to analyze many more iterations until we are sure that the loop has reached a steady state execution. To simplify the problem assume all iterations execute the same as the second iteration of the loop. **You must show your work and explain your calculation.**

— Remember that this microarchitecture is quad-issue! —

`lw x1, 0(x2)`																										
`lw x3, 0(x1)`																										
`mul x4, x3, x5`																										
`addi x2, x2, 4`																										
`addi x6, x6, -1`																										
`sw x4, -4(x2)`																										
`bne x6, x0, loop`																										

Appendix A: TinyRV1 Canonical Microarchitectures

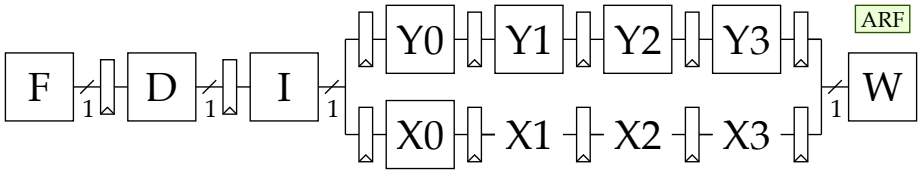

Figure A.1: I3L Microarchitecture for MUL, ADDU, ADDIU

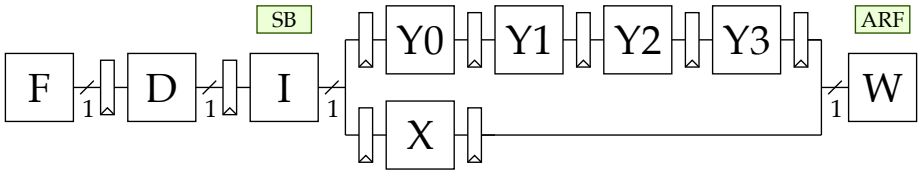

Figure A.2: I2OE Microarchitecture for MUL, ADDU, ADDIU

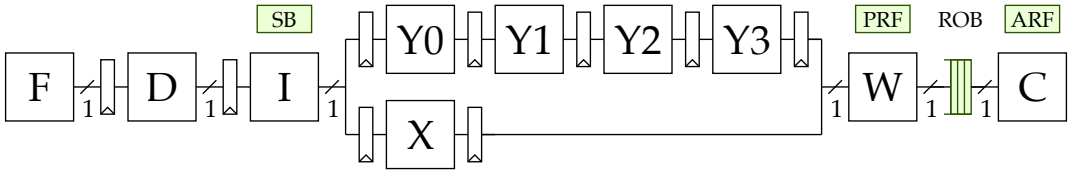

Figure A.3: I2OL Microarchitecture for MUL, ADDU, ADDIU

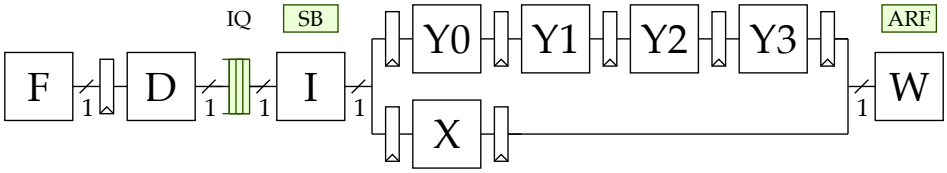

Figure A.4: IO2E Microarchitecture for MUL, ADDU, ADDIU

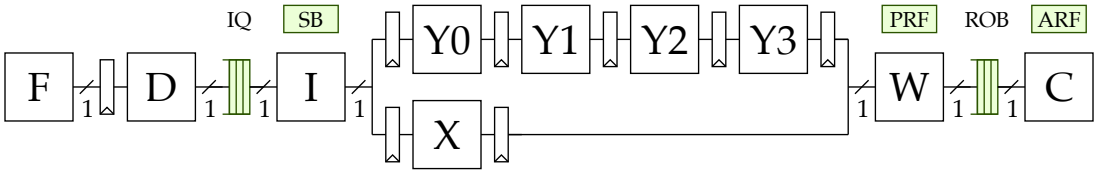

Figure A.5: IO2L Microarchitecture for MUL, ADDU, ADDIU

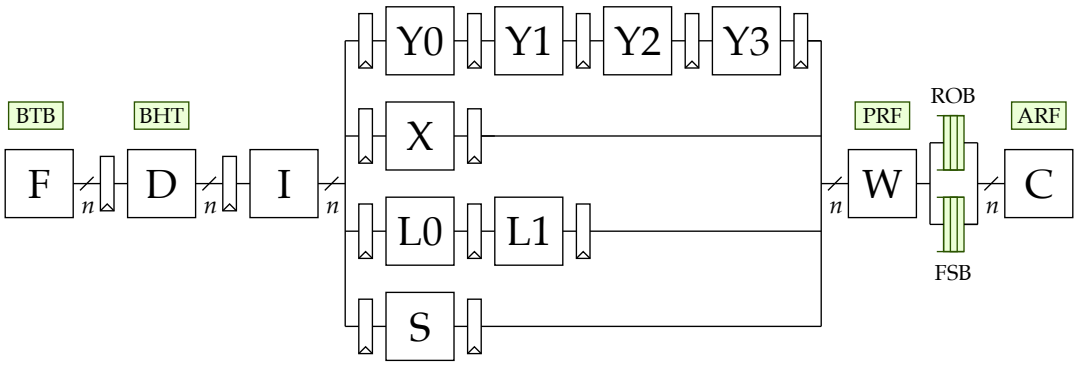

Figure A.6: Complete I2OL Microarchitecture (single issue: n = 1; quad issue: n = 4)

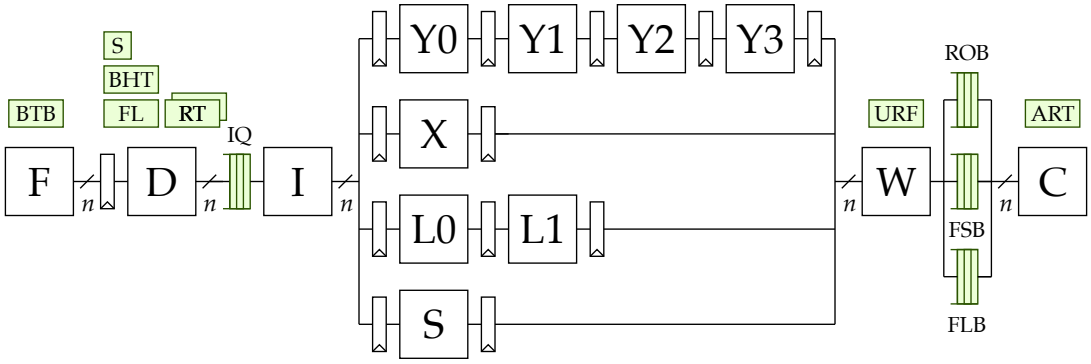

Figure A.7: Complete IO2L Microarchitecture (single issue: n = 1; quad issue: n = 4)