ECE 4750 Computer Architecture

Topic 8: Advanced Processors – Register Renaming

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

revision: 2022-11-30-11-31

List of Problems

Problem 1. Short Answer

Part 1.A Physical Register Deallocation

Assume an instruction *I_x* writes to architectural register *r_a* and that this architectural register is renamed to physical register *p_x*. **Clearly explain when physical register *p_x* is deallocated (and thus can be reused for renaming some other architectural register) for both register renaming schemes discussed in lecture.** Recall that the pointer-based scheme stores pointers in the issue queue and the reorder buffer and the value-based scheme stores values in the issue queue and the reorder buffer.

When can *p_x* be deallocated for pointer-based register renaming?

When can *p_x* be deallocated for value-based register renaming?

Part 1.B Pointer-Based Register Renaming

Consider the complete 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). Assume we use a pointer-based register renaming scheme.

Consider the following assembly instruction sequence.

```
1  sub x1, x2, x3
2  add x4, x5, x6
3  add x4, x4, x3
4  sub x8, x9, x1
5  sub x1, x2, x9
```

Assume that all five instructions are fetched, decoded, and placed into the issue queue, but that no instructions are actually issued from the issue queue. **Show the state of the register rename table and the issue queue after all five instructions are in the issue queue.** The initial state of the register rename table is shown below. The p field is for the pending bit and should be one for pending values and zero if the value is ready. Assume that all registers are initially ready (i.e., not pending) and that free physical registers are allocated in numerical order starting with physical register p9 up to p20.

	p	Physical Reg
x1	0	p0
x2	0	p1
x3	0	p2
x4	0	p3
x5	0	p4
x6	0	p5
x7	0	p6
x8	0	p7
x9	0	p8

Initial State of Register Rename Table

	p	Physical Reg
x1		
x2		
x3		
x4		
x5		
x6		
x7		
x8		
x9		

State of Rename Table after All Instructions Fetched/Decoded

	Op	Dest	p	Src0	p	Src1
0						
1						
2						
3						
4						

State of Issue Queue after All Instructions Fetched/Decoded

Part 1.C Implementing Conditional Moves in IO2L Microarchitecture (Weight ×2)

Consider the complete *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). This microarchitecture includes pointer-based register renaming, memory disambiguation with out-of-order load/store issue, branch prediction, and speculative execution.

In lecture, we discussed conditional moves as a simple form of predication to help turn control flow into data flow. For example, the following conditional move instruction (`movz`) only copies the source register to the destination register if a second source register is zero.

`movz rd, rs1, rs2` if (R[rs1] == 0) R[rd] ← R[rs2]

Study the instruction semantics for `movz` very carefully before continuing. Assume `movz` instructions use the X-pipe and perform the comparison in the X stage. Assume we make any additional modifications to the quad-issue IO2L microarchitecture that are necessary to enable correct execution of `movz` instructions. Consider the following assembly instruction sequence.

```
1  addi x1, x0, 13
2  addi x2, x0, 1
3  mul  x3, x4, x5  # assume R[x4] is 2, R[x5] is 3
4  mul  x3, x3, x6  # assume R[x6] is 3
5  movz x3, x1, x2
6  addi x7, x3, 1
```

What should be the correct value of register x7 after executing this assembly sequence? _____

Draw a pipeline diagram illustrating how this instruction sequence executes on the modified IO2L microarchitecture. Ensure that your pipeline diagram will produce the correct value of register r7 as deteremined above. Use arrows on the pipeline diagram to illustrate RAW dependencies through registers.

— Remember that this microarchitecture is quad-issue! —

`addi x1, x0, 13`																		
`addi x2, x0, 1`																		
`mul  x3, x4, x5`																		
`mul  x3, x3, x6`																		
`movz x3, x1, x2`																		
`addi x7, x3, 1`																		

Problem 2. Register Renaming

In this problem, we will be looking at the detailed microarchitectural state required to implement register renaming both with pointers and with values in the issue queue and reorder buffer. We will be executing the following assembly sequence:

```
1 mul  x1, x2, x3
2 mul  x4, x1, x5
3 add  x6, x7, x8
4 mul  x1, x2, x5
5 add  x6, x6, x9
```

For this problem you should assume that each microarchitecture follows the details described in the lecture notes. Assume a fully-pipelined four-cycle multiplier. Do not assume that all of the instructions are waiting in the issue queue. You must explicitly fetch and decode each instruction in-order.

Part 2.A Architectural RAW, WAW, and WAR Dependencies

List the assembly instructions above and clearly draw arrows to indicate the architectural RAW, WAW, and WAR dependencies between instructions. You must clearly disambiguate the different types of dependencies.

Part 2.B Pipeline Diagram with Register Renaming

Draw the pipeline diagram for the above assembly code with register renaming for an IO2L microarchitecture with an in-order front-end, out-of-order issue and writeback/completion with late commit. Label each column with the cycle number starting with cycle 0.

Part 2.C Register Renaming with Pointers in the IQ/ROB

Create two tables similar to the ones in Figure 1 and 2. Use the tables to show how the microarchitectural state (i.e., the rename table, free list, issue queue, and reorder buffer) changes over time for a register renaming scheme with pointers in the issue queue and reorder buffer. Assume that the physical register file and the architecture register file are combined into a single unified register file (i.e., instead of copying values into a separate architecture register file in commit, we simply update pointers in the architectural rename table). The execution should directly correspond to the pipeline diagram from the previous part. The first three cycles have already been filled in for you. Carefully handle deallocating physical registers and placing them back on the free list.

Fill in the D, I, W, C columns with the instruction number (1–5) to indicate when each instruction is in the corresponding stage. Rename table entries should indicate the correct architectural to physical mapping. Use an asterisk symbol (*) to indicate when a rename table entry is pending. Issue queue entries should be of the form *pdest/psrc0/psrc1* where *pdest* is the physical destination register specifier and *psrc0/psrc1* are the physical source register specifiers. Use an asterisk symbol (*) to indicate when a physical source register specifier is pending (i.e., the source data is not ready in the physical register file yet). Note that to simplify our tables, you only need to show the issue queue entry on the cycle after that entry is written into the issue queue (this allows us to represent the issue queue as a single column). The reorder buffer entries should be of the form *pdest/adest/ppreg*

	Stage				RT										
Cycle	**D**	**I**	**W**	**C**	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**Free List**	**IQ**
0					p0	p1	p2	p3	p4	p5	p6	p7	p8	p9, pA, pB, pC, pD	
1	1				:	:	:	:	:	:	:	:	:	p9, pA, pB, pC, pD	
2	2	1			p9*	:	:	:	:	:	:	:	:	pA, pB, pC, pD	p9/p1/p2
...															

Figure 1: Microarchitectural State (RT/FL/IQ) for Reg Renaming with Pointers in the IQ/ROB

	ROB				
Cycle	**0**	**1**	**2**	**3**	**4**
0					
1					
2	p9*/x1/p0				
...					

Figure 2: Microarchitectural State (ROB) for Reg Renaming with Pointers in the IQ/ROB

where *pdest* is physical destination register specifier, *adest* is the architectural destination register specifier, and *ppreg* is the previous physical register specifier. Use an asterisk (*) symbol to indicate when an entry in the ROB is not finished. You can use a vertical line (|) or vertical dots (:) to indicate that a field does not change on that cycle. Note that the physical register specifiers are denoted in hex from p0 to pF. *All state should be shown as in the table as it exists at the beginning of the cycle.*

Part 2.D Register Renaming with Values in the IQ/ROB

Create a table similar to the one in Figure 3. Use the table to show how the microarchitectural state (i.e., RT, IQ, ROB) changes over time for a register renaming scheme with values in the issue queue and the reorder buffer. The execution should directly correspond to the pipeline diagram from the previous part. The first column and first three cycles have already been filled in for you. Carefully handle deallocating physical registers and placing them back on the free list.

Fill in the D, I, W, C columns with the instruction number (1–5) to indicate when each instruction is in the corresponding stage. Renaming table entries should indicate the correct architectural to physical mapping. If there is no mapping in the rename table (i.e., that entry is invalid), then we get the value from the architectural register file. Use an asterisk symbol (*) to indicate when a rename table entry is pending. Issue queue entries should be of the form *pdest/src0/src1* where *pdest* is the physical destination register specifier and *src0/src1* are either the physical source register specifiers or the actual source data. Use the architectural register specifier for the source in the table if the data is read from the architectural register file and copy it into the issue queue in the decode stage. Use the physical register specifier if the source is not ready yet. For consistency, use an asterisk symbol (*) to indicate when a physical source register specifier is pending (i.e., the source data is being written by an instruction in-flight that has not finished yet). Note that to simplify our tables,

	Stage				RT										ROB				
Cycle	**D**	**I**	**W**	**C**	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**IQ**	**0**	**1**	**2**	**3**	**4**
0																			
1	1																		
2	2	1			p0*									p0/x2/x3	p0*/x1				
...																			

Figure 3: Microarchitectural State for Reg Renaming with Values in the IQ/ROB

you only need to show the issue queue entry on the cycle after that entry is written into the issue queue (this allows us to represent the issue queue as a single column). The reorder buffer entries should be of the form *pdest/adest* where *pdest* is physical destination register specifier (this is always the same as the ROB entry number) and *adest* is the architectural destination register specifier. You can use a vertical line (|) or vertical dots (:) to indicate that a field does not change on that cycle. *All state should be shown as in the table as it exists at the beginning of the cycle.*

Appendix A: TinyRV1 Canonical Microarchitectures

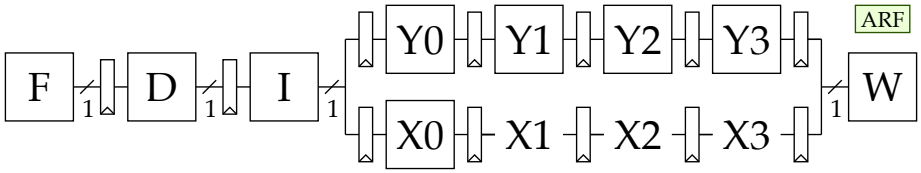

Figure A.1: I3L Microarchitecture for MUL, ADDU, ADDIU

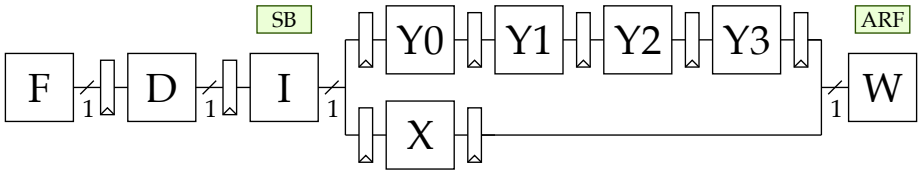

Figure A.2: I2OE Microarchitecture for MUL, ADDU, ADDIU

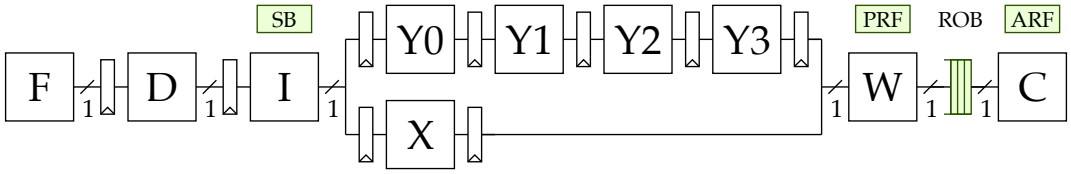

Figure A.3: I2OL Microarchitecture for MUL, ADDU, ADDIU

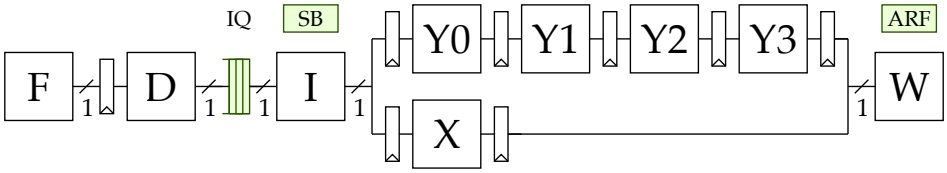

Figure A.4: IO2E Microarchitecture for MUL, ADDU, ADDIU

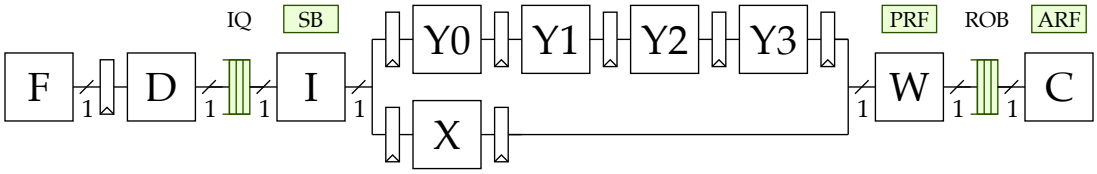

Figure A.5: IO2L Microarchitecture for MUL, ADDU, ADDIU

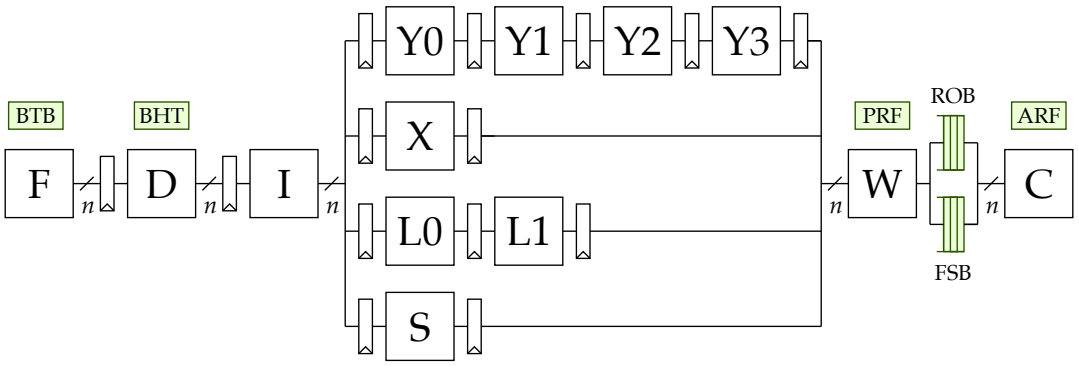

Figure A.6: Complete I2OL Microarchitecture (single issue: n = 1; quad issue: n = 4)

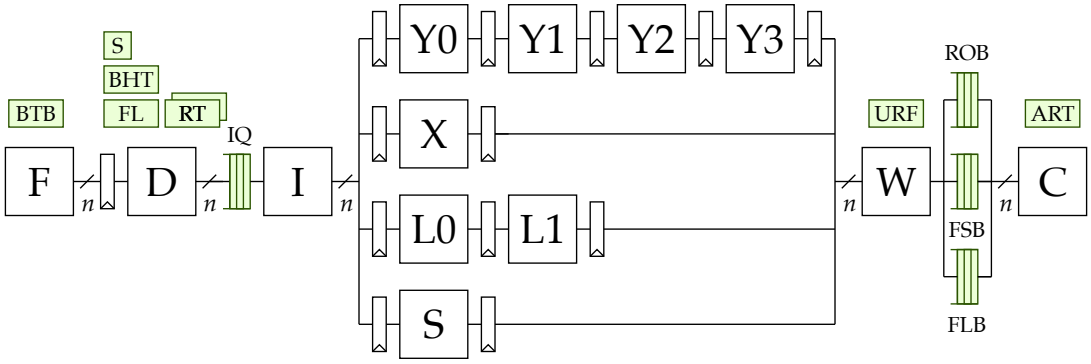

Figure A.7: Complete IO2L Microarchitecture (single issue: n = 1; quad issue: n = 4)