CS152 Midterm 2 Review Solutions

Out-of-Order Processors

Consider two processor pipelines.Processor A is an out-of-order, dual-issue superscalar that uses a typical Unified Physical Register File scheme. Processor B is also a dual-issue out-of-order pipeline, but does not support any type of register renaming.

A) For which programs(s) will Processor A have fewer bubbles than Processor B? Why?

`// Program 1`	`// Program 2`	`// Program 3`
`lw    t0, 0(t2)`	`lw   t0, 0(t2)`	`lw   t0, 0(t2)`
`addi  t1, t1, 1`	`add  t1, t0, t1`	`addi  t1, t0, 1`
`addi  t2, t2, 1`	`addi t0, t4, 1`	`addi  t2, t1, 1`
`addi  t3, t3, 1`	`sw   t0, 0(t5)`	`addi  t3, t2, 1`
`beq   t0, t4, done`	`beq  t1, t4, done`	`beq   t3, t4, done`

Program 2 will benefit the most from register renaming, as it has Write After Write hazard on t0. Therefore, it will have fewer bubbles on Processor A than on Processor B.

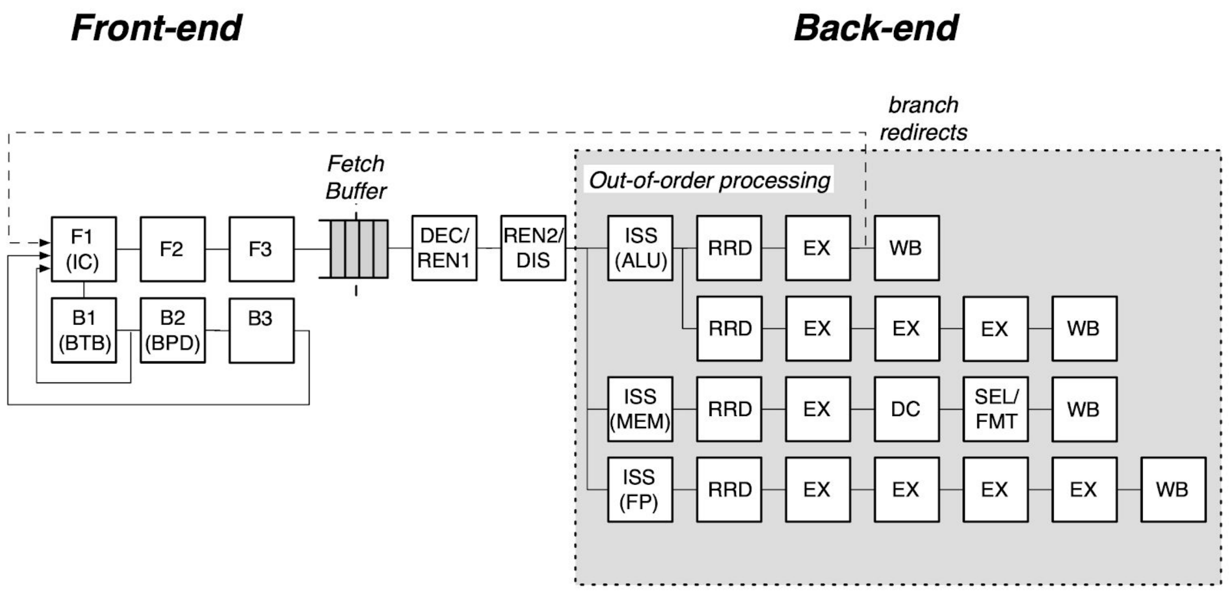
In contrast, Program 3 has a chain of true dependencies or Read After Write hazards, so it will have minimal instruction-level parallel on both processors.

B) How does adding register renaming affect the Instructions / Program term of the Iron Law? The Cycles / Instruction term?

Adding register renaming reduces Cycles per Instruction, as it prevents the processor from having to stall when an instruction overwrites an architectural register that is still being used by an earlier instruction. It has no effect on the number of instructions per program.

Out-of-Order Pipeline Latency Diagram

(to be used for the rest of this question)



Use the following information to determine how execution progresses

- The machine can fetch, dispatch, and issue **one** instruction per cycle
- The processor runs the RISC-V instruction set with the floating point extension
- Assume every load hits in the single-cycle-hit L1 D$ (indicated as DC in the pipeline)
- Register renaming follows the **Unified Physical Register File** scheme
- Unless otherwise directed, **assume there are no bypass paths for data**
- Instructions are written into the ROB at the end of the DEC/REN1 stage
- Instructions are written into the issue queue at the end of the REN2/DIS stage
- Instructions are read from the issue queue in the ISS stage. When an instruction is selected for issue in a given cycle, it is in the ISS stage for that cycle and is said to "issue" that cycle.
- Instructions write their results to PRd at the end of the WB stage. This is also when they set the `done` bit in their ROB entry. An instruction "completes" in the writeback stage.
- Commit is handled by a decoupled unit that looks at the ROB entries. At most one instruction may commit per cycle. Registers appear on the free list at the end of the "commit" cycle.
- Jump instructions complete on the same cycle that they dispatch. They do not go to an issue queue and do not use an issue slot. Assume all jump targets are perfectly predicted.

C) Consider the following instruction sequence:

```
A: add  t3, t2, t1
B: add  t4, t3, t3
```

Assume that instruction A is in the WB stage during cycle 6. On what cycle is it theoretically safe to issue instruction B? Why is this the case? What will the issue logic have to do to accommodate this?

If instruction A is in the WB stage on cycle 6, it will write back its results to the physical register file at the end of cycle 6. Since the RRD stage is one later than the ISS stage, an instruction that issues on cycle 6 will not read its operand registers until cycle 7. Therefore, it is theoretically safe to issue instruction B on cycle 6. In order to accomplish this, the issue logic would have to combinatorially receive the "wakeup" signal from the completion of instruction A. Alternately, instructions could broadcast the present bit on their results one stage before WB. Note that these are just microarchitectural implementation strategies that are tied to the details of this question.

D) What is the absolute minimum number of physical registers that a unified physical register file, out-of-order RISC-V machine could have and still work? Justify your response.

Note: this does not discuss floating point; we will be more careful to be clear on the exam.

Good answer: A unified physical register file machine must have enough physical registers to hold committed copies of all the architectural registers. Furthermore, it needs one extra physical register to assign as PRd to an instruction when it enters the ROB. Therefore, it needs 32 + 1 = 33 physical registers.

Slightly better answer: A unified physical register file machine must have enough physical registers to hold committed copies of all the architectural registers. Furthermore, it needs one extra physical register to assign as PRd to an instruction when it enters the ROB. However, the handling of r0 can be optimized to avoid assigning a physical register. Therefore, it needs 31 + 1 = 32 physical registers.

F) Consider the following code sequence that begins at address 0x80001104

```
loop:
       fld    f0, 0(t1)
       fld    f1, 0(t2)
       fmul.d f0, f0, f1
       fadd.d f2, f2, f0
       fld    f0, 8(t2)
       fadd.d f2, f2, f0
       j      loop
```

Start from cycle 0, in which:

- The rename table and free list have the following initial state
- The first fld is in the REN2/DIS stage, having just added its entry to the ROB
- No other entries are in the ROB
- All instructions have already been fetched into the fetch buffer already be fetched, assuming perfect jump target prediction.

Unused architectural registers are omitted from the rename table for clarity.

Rename table			**Free list (dequeue from top):**
t0	p5		~~p13~~ p8 p6 p18 p7 p4 p1 p9 p12
t1	p3		
t2	p11		
f0	~~p10~~ p13		
f1	p19		
f2	p14		

Fill in the following table (which describes the execution of each instruction) for seven instructions, beginning with the first `fld`. In the "Time" columns, fill in the cycles in which the instruction dispatches, issues, completes, and commits, respectively. You should use the tables on the previous page to help keep track of the state of the machine, but they will not be graded. Assume none of the instructions cause exceptions, and that the issue queues are never full.

Complete every blank box in the table. Assume the fast issue logic from part (C)

	Physical Registers				**Cycle #**			
PC	**PRd**	**LPRd**	**PR1**	**PR2**	**Disp.**	**Issue**	**Comp.**	**Commit**
0x80001104	p13	p10	p3	--	0	1	6	7
0x80001108	p8	p19	p11	--	1	2	7	8
0x8000110C	p6	p13	p13	p8	2	7	13	14
0x80001110	p18	p14	p14	p6	3	13	19	20
0x80001114	p7	p6	p11	--	4	5	10	21
0x80001118	p4	p18	p18	p7	5	19	25	26
0x8000111C	--	--	--	--	6			27
0x80001104	p1	p7	p3	--	7	8	13	28
0x80001108	p9	p8	p11	--	8	9	14	29
0x8000110C	p12	p1	p1	p9	9	14	20	30

How many physical registers does this machine have? What would happen if there were [FOUR] fewer physical registers?

When the ROB is empty, there are 9 registers on the free list. Assuming r0 is optimized away, there are also physical registers holding 31 integer and 32 FP architectural registers. Therefore, there are 9 + 31 + 32 = 72 physical registers.

Without the assumption of optimized handling of r0, there would be 73 physical registers.

If there were [FOUR] fewer registers, the machine would have to stall while the free list is empty. It would not be able to put the last instruction in the table into the ROB until after cycle 20, meaning it would not dispatch until one or more cycle later (commit to free list latency being unspecified).

VLIW Machines

A) What is the defining characteristic of a software-pipelined implementation of a loop?

A software-pipelined implementation will execute multiple different iterations of the loop in at the same time by scheduling different operations from each iteration in parallel.

B) Assume that register t3 starts with value 0x4. What is its value of address 0x48 after the following sequence of VLIW instructions? Is the branch taken?

Int1	**Int2**	**Mem**
`add   t4, t3, t3`	`addi  t3, t3, 0x2`	
`add   t4, t3, t3`	`addi  t3, t4, 0x0`	
`beq   t4, t3, done`	`addi  t4, t4, 0x1`	`sw t4, 0x48(r0)`

After the first instruction:	t4 = 0x8, t3 = 0x6
After the second instruction:	t4 = 0xC, t3 = 0x8

Address 0x48 gets the value 0xC, and the branch is not taken.

C) Consider the following naive code for a strcpy routine:

```
strcpy:    lb   t0, 0(a0)
           sb   t0, 0(a1)
           addi a0, a0, 0x1
           addi a1, a1, 0x1
           bne  t0, r0, strcpy
done:
```

Assuming cache hits take >1 cycle, optimize the code to improve CPI without unrolling

```
strcpy:    lb   t0, 0(a0)
           addi a0, a0, 0x1
           addi a1, a1, 0x1
           sb   t0, -1(a1)
           bne  t0, r0, strcpy
```

D) Manually unroll the code to do two iterations per backwards jump.

```
strcpy:    lb   t0, 0(a0)
           lb   t1, 1(a0)
           addi a0, a0, 0x2
           addi a1, a1, 0x2
           sb   t0, -2(a1)
           beq  t0, r0, done
           sb   t1, -1(a1)
           bne  t1, r0, strcpy
done:
```

E) Complete the following software-pipelined, unrolled, VLIW implementation. All branches must go in the 'Int1' execution slot. The prologue has been completed for you.

Label	Int1	Int2	Load	Store
`strcpy:`		`addi a0, a0, 2`	`lb t0, 0(a0)`	
		`addi a1, a1, 2`	`lb t1, -1(a0)`	
`loop:`	`beq t0, r0, done`	`addi a0, a0, 0x2`	`lb  t0, 0(a0)`	`sb  t0, -2(a1)`
	`bne t1, r0, loop`	`addi a1, a1, 0x2`	`lb  t1, -1(a0)`	`sb  t1, -1(a1)`
`done:`				

F) What is the maximum allowable latency of a load that still allows the loop to run without stalls? Assume all accesses hit in the L1 D$.

2 cycles

Branch History Tables

Consider the following C loop, which accumulates a vector sum.

```
for (i = 0; i < n; i++)
  sum += a[i]
```

It translates to the following assembly, and is run on a processor with a 512-entry BHT.

```
           addi t0, r0, 0x0   // PC = 0x81001000
loop:      beq  t0, a1, done // BHT index 1 weak not take
           lw   t1, 0(a0)
           addi a0, a0, 0x4
           addi t0, t0, 0x1
           add  t2, t2, t1
           j    loop         //
done:
```

After this code runs for n > 10, is it possible to know the final value of any BHT entries? If so, list the index and associated value for each.

Index 1 has value “weak taken,” which would correspond to 0x2, if 0x3 is “strong taken.”