Final Exam Review: Part 1, 5/3/19

Iron Law

	Instructions / Program	Cycles / Instruction	Seconds / Cycle	Execution Time
Pipelining a single-cycle implementation				
Adding stages to an existing pipeline				
Adding bypass paths				
Adding hardware floating-point instructions				

Data-in-ROB Machines

Consider a dual-issue core with a data-in-ROB design. The ROB has twelve entries. Instructions write back the same cycle they complete, and can commit one cycle later. ROB entries can be reused one cycle after commit. Instructions can issue on the same cycle that the instruction(s) they depend on write back. Loads and stores take three cycles each, ALU instructions take one cycle, and branches resolve / complete using the ALU one cycle after they issue. All functional units are fully pipelined.

Fill out the table with the cycles at which instructions enter the ROB, issue to the functional units, complete, and commit, and record all destination and operand names. Use ROB0-ROB11 for the twelve ROB entries. If the instruction producing a source register has committed before the dependent instruction enters the ROB, use the architectural register name. On each cycle, two instructions can enter the ROB, and one instruction **of each type** can issue and complete. Up to two instructions of any type may commit per cycle.

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

Fill out the tables below for executing the above strcpy routine the string “H” as input. Assume the branch is always predicted taken in time to fill the fetch buffer, and that the ROB is empty before the first load. Assume older instructions are chosen to issue first. Fill the table out through when the mispredicted branch is caught. What happens then? Circle the event in the table that corresponds with the earliest time when this would be corrected, assuming branch mispredicts are handled more quickly than exceptions.

	Cycle #				**Data Location**		
Instruction	**Enter ROB**	**Issue**	**Complete**	**Commit**	**Dest**	**Src1**	**Src2**
`lb t0, 0(a0)`	0				ROB0	a0	--
`sb t0, 0(a0)`	0						

Trace Scheduling

Trace scheduling is a compiler technique that increases ILP by removing control dependencies, allowing operations following branches to be moved up and speculatively executed in parallel with operations before the branch. It was originally developed for statically scheduled VLIW machines, but it is a general technique that can be used in different types of machines, and in this question we apply it to a single-issue RISC-V processor.

Consider the following RISC-V code sequence:

```
B1:
    fdiv.d f1, f2, f3
    fadd.d f4, f1, f5
    beqz x1, B3        # Taken 99%
B2:
    ld x2, 4(x3)
    j B4
B3:
    ld x2, 0(x3)
B4:
    addi x2, x2, 8
    beqz x2, B6        # Taken 99%
B5:
    fsub.d f2, f3, f7
    j B7
B6:
    fsub.d f2, f2, f6
    sd f2, 0(x8)
B7:
    addi x3, x3, 8
    addi x8, x8, 8
```

The code is executed on an in-order single-issue RISC-V pipeline. Integer arithmetic instructions are fully pipelined with a single-cycle latency. Loads are fully pipelined with a two-cycle latency. Floating-point add and subtract instructions are fully pipelined with a three-cycle latency. Floating-point divide instructions are unpipelined with an 8-cycle latency, but other independent instructions can execute while the divider is busy.

Branches that are not taken execute in a single cycle. Taken branches and unconditional jumps incur two stall cycles (three cycles total).

Part A: Assume both conditional branches are taken and that all register values are available on the first cycle. How long does the code sequence take to execute (i.e., total pipeline occupancy)?

Part B: Consider only the code along the most frequently taken trace. Omit the branches, and show how to reschedule the code along this trace to execute in the least number of cycles, without modifying load or store offsets. How many cycles does this trace take?

Part C: Add branches to correctly exit the trace on the infrequent paths and show the fixup code required on these exits, without modifying load/store offsets. Your solution should minimize the slowdown to the most commonly followed trace. How many cycles does this hot trace now take?

Vector ISAs

Vectorize the following double-precision dot product C code using the RVV vector ISA described in Appendix A. Your code should perform well for vectors of >10000 elements.

```
double ddot(int n, double *x, double *y) {
  double result = 0.0;
  for (int i = 0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

Part A: Vectorize the code. Assume that register `a0` holds `n`, register `a1` holds `x`, and register `a2` holds `y`. Return the result in register `a0`. You may reorder the floating-point arithmetic operations to improve efficiency. As a simplifying assumption, assume that `N` is evenly divisible by the maximum vector length `MVL`.

```
done:     ret
```

Part B: Discuss at least two ways we could modify this code to support vectors that have lengths not evenly divisible by MVL.

Appendix A: Vector Architecture for Question 1

This instruction listing is identical to lab 4's but with a setvl instruction that has identical semantics to the preprocessor macro provided in lab 4. This instruction first sets VL to *min(maximum vector length, rs1);* and then returns the new VL. Omitting the final vector mask (`vm`) argument to all instructions is legal, and treats all elements $i < VL$ as active.

Instruction	Operation
`setvl rd, rs1`	`VL := min(MVL, rs1); (rd) := VL`
`vld vd, offset(rs1), vm`	`vd[i] := mem[(rs1) + offset + i]`
`vst vs3, offset(rs1), vm`	`mem[(rs1) + offset + i] := vs3[i]`
`vlds vd, offset(rs1), rs2, vm`	`vd[i] := mem[(rs1) + offset + i * rs2]`
`vsts vs3, offset(rs1), rs2, vm`	`mem[rs1 + offset + i * (rs2)] := vs3[i]`
`vldx vd, offset(rs1), vs2, vm`	`vd[i] := mem[(rs1) + offset + vs2[i]]`
`vstx vs3, offset(rs1), vs2, vm`	`mem[(rs1) + offset + vs2[i]] := vs3[i]`
`vadd vd, vs1, vs2, vm`	`vd[i] := vs1[i] + vs2[i]`
`vsub vd, vs1, vs2, vm`	`vd[i] := vs1[i] - vs2[i]`
`vmul vd, vs1, vs2, vm`	`vd[i] := vs1[i] * vs2[i]`
`vdiv vd, vs1, vs2, vm`	`vd[i] := vs1[i] / vs2[i]`
`vrem vd, vs1, vs2, vm`	`vd[i] := vs1[i] % vs2[i]`
`vmax vd, vs1, vs2, vm`	`vd[i] := max(vs1[i], vs2[i])`
`vmin vd, vs1, vs2, vm`	`vd[i] := min(vs1[i], vs2[i])`
`vsl vd, vs1, vs2, vm`	`vd[i] := vs1[i] << vs2[i]`
`vsr vd, vs1, vs2, vm`	`vd[i] := vs1[i] >> vs2[i]`
`vseq vd, vs1, vs2, vm`	`vd[i] := vs1[i] == vs2[i] ? 1 : 0`
`vsne vd, vs1, vs2, vm`	`vd[i] := vs1[i] != vs2[i] ? 1 : 0`
`vslt vd, vs1, vs2, vm`	`vd[i] := vs1[i] < vs2[i] ? 1 : 0`
`vsge vd, vs1, vs2, vm`	`vd[i] := vs1[i] >= vs2[i] ? 1 : 0`
`vaddi vd, vs1, imm, vm`	`vd[i] := vs1[i] + imm`
`vsli vd, vs1, imm, vm`	`vd[i] := vs1[i] << imm`
`vsri vd, vs1, imm, vm`	`vd[i] := vs1[i] >> imm`
`vmadd vd, vs1, vs2, vs3, vm`	`vd[i] := vs1[i] * vs2[i] + vs3[i]`
`vmsub vd, vs1, vs2, vs3, vm`	`vd[i] := vs1[i] * vs2[i] - vs3[i]`
`vnmadd vd, vs1, vs2, vs3, vm`	`vd[i] := -(vs1[i] * vs2[i] + vs3[i])`
`vnmsub vd, vs1, vs2, vs3, vm`	`vd[i] := -(vs1[i] * vs2[i] - vs3[i])`
`vslide vd, vs1, rs2, vm`	`vd[i] := 0 ≤ (rs2) + i < VL ? vs1[(rs2) + i] : 0`
`vinsert vd, vs1, rs2, vm`	`vd[(rs2)] := (rs1)`
`vextract rd, vs1, rs2, vm`	`(rd) := vs1[(rs2)]`
`vmfirst rd, vs1`	`(rd) := ([i for i in range(0, VL)` `if LSB(vs1[i]) == 1] + [-1])[0]`
`vmpop rd, vs1`	`(rd) := len([i for i in range(0, VL)` `if LSB(vs1[i]) == 1])`
`vselect vd, vs1, vs2, vm`	`vd[i] := vs2[i] < VL ? vs1[vs2[i]] : 0`
`vmerge vd, vs1, vs2, vm`	`vd[i] := LSB(vm[i]) ? vs2[i] : vs1[i]`