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

SOLUTIONS v1.2

Iron Law

Do not write "sec/cycle goes up because more hardware" or similar!

	Instructions / Program	Cycles / Instruction	Seconds / Cycle	Execution Time
Pipelining a single-cycle implementation	No change This is a purely microarchitectural change	Increase A fully-pipelined implementation will still have CPI > 1	Decrease Pipelining decreases the number of serial levels of logic that data must propagate through in one cycle	Most likely decrease Almost any design can benefit from some degree of pipelining
Adding stages to an existing pipeline	No change This is a purely microarchitectural change	Increase Deeper pipelines generally are more susceptible to hazards	Decrease Pipelining decreases the number of serial levels of logic that data must propagate through in one cycle	Ambiguous As pipelines get "too deep," the CPI increase outweighs the diminishing clock rate improvement
Adding bypass paths to a 5-stage pipeline	No change This is a purely microarchitectural change	Decrease Bypass paths resolve hazards, allowing instructions to avoid stalls	Assuming that the bypass paths are comparable with the reasonable ones from a 5-stage pipeline, they take advantage of slack in timing, and have no effect on seconds/cycle **OR** Increase, as they may add to the depth of serial logic in long pipeline stages	Decrease OR Ambiguous, given that they could combine stages entirely (e.g., bypassing load data output to input of ALU). A good justification with concrete reasoning is necessary.

Adding hardware floating-point instructions	Decrease Software floating-point routines can be replaced by single instructions	Increase Floating point instructions likely have multi-cycle latencies; the presence of dependencies will prevent this from always being hidden by pipelining	No change The longer execution of floating-point instructions is almost always handled by pipelining the FPU	Decrease Floating point is commonly used, and any program using it will purely benefit

Data-in-ROB Machines

Consider a dual-issue Out-of-Order 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 prioritized for issue. Fill the table out through when the mispredicted branch is caught. What happens? Circle the table entry corresponding with the earliest time this could be corrected, assuming branch mispredicts are handled more quickly than exceptions. For an infinite string, what is the limit on CPI for this loop?

	Cycle #				Data Location		
Instruction	**Enter ROB**	**Issue**	**Complete**	**Commit**	**ROB Slot**	**Src1**	**Src2**
`lb t0, 0(a0)`	0	1	4	5	ROB0	a0	--
`sb t0, 0(a1)`	0	4	7	8	ROB1	a1	ROB0
`addi a0, a0, 0x1`	1	2	3	8	ROB2	a0	--
`addi a1, a1, 0x1`	1	3	4	9	ROB3	a1	--
`bne t0, r0, loop`	2	4	5	9	ROB4	ROB0	r0
`lb t0, 0(a0)`	2	3	6	10	ROB5	ROB2	--
`sb t0, 0(a1)`	3	6	9	10	ROB6	ROB3	ROB5
`addi a0, a0, 0x1`	3	5	6	11	ROB7	ROB2	--
`addi a1, a1, 0x1`	4	6	7	11	ROB8	ROB3	--
`bne t0, r0, loop`	4	7	8	12	ROB9	ROB5	r0
`lb t0, 0(a0)`	5	7	10	--	ROB10	ROB7	--
`sb t0, 0(a1)`	5	--	--	--	ROB11	ROB8	ROB10
`addi a0, a0, 0x1`	6	8	9	--	ROB0	ROB7	--
(nothing else can enter ROB before cycle 9)							

- When the mispredicted branch is caught, the ROB entries after the branch are flushed, and the correct branch direction (not taken) is fed back to the fetch stage.

- The earliest that the mispredict could be corrected is the cycle where the mispredicted branch completes.

- How the table is adjusted to correct for branch misprediction recovery isn't as precisely defined as it would be for a midterm question, since the table itself is not a piece of microarchitectural state. Don't worry if the last four rows of your table differ in which fields are blanked out.

- The branch does commit.

- The "limit on CPI" question is a bit too much of an add-on to this question, as it requires you to analyze what the table would have looked like if the string had continued. In the infinite string case, the next `addi` that would have occupied the last line in the table cannot enter the ROB until cycle 9, as the first `sb` instruction does not commit until cycle 8. This pattern of stalling on ROB entries for multiple cycles repeats in the steady state, and the number of ROB slots therefore is a limit that increases CPI.

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)?

```
 1 fdiv.d f1, f2, f3
 9 fadd.d f4, f1, f5
10 beqz x1, B3
13 ld x2, 0(x3)
15 addi x2, x2, 8
16 beqz x2, B6
19 fsub.d f2, f2, f6
22 sd f2, 0(x8)
23 addi x3, x3, 8
24 addi x8, x8, 8
25 …
```

First cycle of following code - first cycle of code sequence = "how long code takes to execute"

25 - 1 = 24 cycles

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?

```
 1 fdiv.d f1, f2, f3
 2 ld x2, 0(x3)
 3 fsub.d f2, f2, f6
 4 addi x2, x2, 8
 5 addi x3, x3, 8
 6 sd f2, 0(x8)
 7 addi x8, x8, 8
 9 fadd.d f4, f1, f5
10 ...
```

10 - 1 = 9 cycles

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?

There are multiple possible solutions. With some justification, it would also be possible to claim that the `B2` and `B5` blocks could be stored at some far-off location in the text section, rather than immediately after the most frequent (hot) trace. This would allow the code that follows this sequence to appear immediately after the hot trace, eliminating the need for the `j end` instruction.

```
        fdiv.d f1, f2, f3     # 1
        bnez x1, B2           # 2
        ld x2, 0(x3)          # 3
        addi x2, x2, 8        # 5
cont:   bnez x2, B5           # 6
        fsub.d f2, f2, f6     # 7
        addi x3, x3, 8        # 8
        sd f2, 0(x8)          # 9
        addi x8, x8, 8        # 10
        fadd.d f4, f1, f5     # 11
        j end                 # 12

 B2:    ld x2, 4(x3)
        j cont

 B5:
        fsub.d f2, f3, f7
        addi x2, x2, 8
        addi x3, x3, 8
        addi x8, x8, 8
        fadd.d f4, f1, f5

end:    ...                   # 15
```

15 - 1 = 14 cycles

By having the code “fall through” to the next block and relocating B2 and B5 to another area in the text section, it could be reduced to 11 cycles.

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`.

In general, it is possible to make the reduction cheaper by using a tree-like pattern. This requires some assumptions on what the legal values of MVL are, and wasn't necessary for a satisfactory answer to this question. Note that the floating-point result of the dot product is held in `a0`, a scalar register. It would need an conversion to be used later on in the program.

```
        setvl   t0, a0          # VL = MVL
        slli    t2, t0, 3       # t2 = VL * 8
        vslide  v0, v1, t0      # zero out v0
loop:   vld     v1, 0(a1)
        vld     v2, 0(a2)
        vmadd   v0, v1, v2, v0  # v0 += v1*v2
        sub     a0, a0, t0
        addi    a1, a1, t2
        add     a2, a2, t2
        bne     a0, r0, loop
sum:    addi    t1, r0, 1
        vslide  v1, v0, t1      # v1[i]=v0[i+1]
accum:  vadd    v0, v0, v1      # accumulated
        vslide  v1, v1, t1      # slide v1 left one
        addi    t0, t0, -1      # decrement counter
        bne     t0, t1, sum     # only do VL-1 iterations
val:    vextract a0, v0, r0     # put result in int register a0
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.

- Add scalar code to handle the remainder of the dot product
- Use predication

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]`