CS 152: Computer Architecture and Engineering
CS 252: Graduate Computer Architecture

Midterm #2
April 17th, 2019
Professor Krste Asanović

SOLUTIONS

Name:________________________

SID:________________________

I am taking CS152 / CS252

(circle one)

This is a closed book, closed notes exam.
80 Minutes, 19 pages.

Notes:

- Not all questions are of equal difficulty, so look over the entire exam!
- Please carefully state any assumptions you make.
- Please write your name on every page in the exam.
- Do not discuss the exam with other students who haven't taken the exam.
- If you have inadvertently been exposed to an exam prior to taking it, you must tell the instructor or TA.
- You will receive no credit for selecting multiple-choice answers without giving explanations if the instructions ask you to explain your choice.

Question	CS152 Point Value	CS252 Point Value
1	20	16
2	20	--
3	20	20
4	20	20
Grad Supplement	--	20
TOTAL	80	76

Problem 1: Vector Machines and Company

Multiple choice: Check one unless otherwise noted

A) **(1 Point)** What sort of parallelism do vector machines primarily exploit?

[ ] ILP [X] DLP [ ] TLP

B) **(1 Point)** What architectural features to exploit parallelism are present in a modern, general-purpose processor (e.g. x86 server processor) (check all that apply).

[X] SIMD [X] Multi-threading [X] Superscalar Execution [ ] VLIW [X] Pipelining

C) **(1 Point)** Which technique do both GPUs and vector machines use to remove per-element control hazards?

[X] Predication [ ] Trace Scheduling [ ] Branch Prediction

D) **(3 Points) Short Answer:** Give one distinguishing feature of a traditional vector architecture (e.g. Cray-style vectors) versus a packed-SIMD architecture (e.g. Intel AVX)? Give one advantage of each approach.

Specifically, we were looking for *variable vector length*. We accepted a few others too.

Vector advantages:

- Expressive instructions make it easier to write vectorized assembly for a wider range of applications.
- Code is more portable, as VL can be set up to the maximum vector length of the machine
- More efficient ISA encoding, since opcode space isn't wasted defining multiple variants of instructions acting on different vector lengths.

SIMD advantages:

- Classical (subword) packed-SIMD machines are simple to implement in hardware and can reuse GPRs.

It's important to note that that contemporary packed-SIMD architectures (e.g. Intel AVX) look increasingly like traditional vector machines (support for scatter-gather, strided memory ops, much longer vectors, predication) but still lack support for variable length.

E) **(12 points for 152, 8 points for 252)** Vectorize the following **double-precision** C code using the RISC-V vector specification described in lab 4. See appendix A for the vector instruction set listing.

```
for (i = 0; i < N; i++) {
    D[i] = A[i] + B[i] * C[i];
}
```

Assume:

- Vector registers v0 – v8 have been configured to hold vectors of double-precision floats.
- Register a0 holds an integer N; `a1 – a4` hold `double*` A, B, C and D, respectively.
- A, B, C, and D do not overlap.
- Feel free to use registers `a5 – a7` to hold scalar values

```
# There are myriad legal code sequences; this is one
stripmine_loop:
      # Your code begins
      setvl a5, a0
      # Load input vectors
      vld v1, 0(a1)
      vld v2, 0(a2)
      vld v3, 0(a3)
      # B[] * C[] + A[]
      vmadd v4, v2, v3, v1
      # Store back to D
      vst v4, 0(a4)
      # Subtract the vector length from the input length
      sub a0, a0, a5
      # Calculate the pointer bump
      slli a6, a5, 3
      # Bump all pointers
      add a1, a1, a6
      add a2, a2, a6
      add a3, a3, a6
      add a4, a4, a6
      # Branch if there are still elements to process
      bne a0, zero, stripmine_loop
      # Your code ends
```

F) **(2 points)** Name a vector-specific microarchitectural technique one could apply to improve throughput on the code above.

Chaining

Problem 2: VLIW

In this problem, we will optimize a vector-vector add kernel for a VLIW machine.

```
// C implementation
void vvadd(double restrict *A,
           double restrict *B,
           double restrict *C,
           int n)
{
  for (int i = 0; i < n; i++)
    C[i] = A[i] + B[i];
}
```

```
# Naive RISC-V implementation
# t0: i, a0: A, a1: B, a2: C, a3: n
# Assume n > 0, t0 = 0
       ...
loop:  fld    f0, 0(a0)
       fld    f1, 0(a1)
       fadd.d f0, f0, f1
       fsd    f0, 0(a2)
       addi   a0, a0, 0x8
       addi   a1, a1, 0x8
       addi   a2, a2, 0x8
       addi   t0, t0, 0x1
       bne    t0, a3, loop
done:  jr     ra
```

The program will be mapped to a VLIW machine with the following specs:

- Two ALU units with one-cycle latency; ALU1 is used for branches
- One fully-pipelined load unit with a two-cycle latency
- One fully-pipelined store unit. For this question, ignore the latency of memory-memory dependencies and assume `C` does not overlap with `A` or `B`.
- One fully-pipelined FPU with a three-latency
- **There are no interlocks, and all latencies are explicitly exposed in the ISA**

Assumptions:

- Register t0 is initialized to zero before the start of your code, and n > 0.
- There are no exceptions or interrupts in the execution of the program.
- **You may assume that n is a very large number.**

A) (8 points, 152 ONLY) Schedule the algorithm on the VLIW machine without unrolling or software pipelining. Try to minimize the number of cycles, but prioritize correctness!

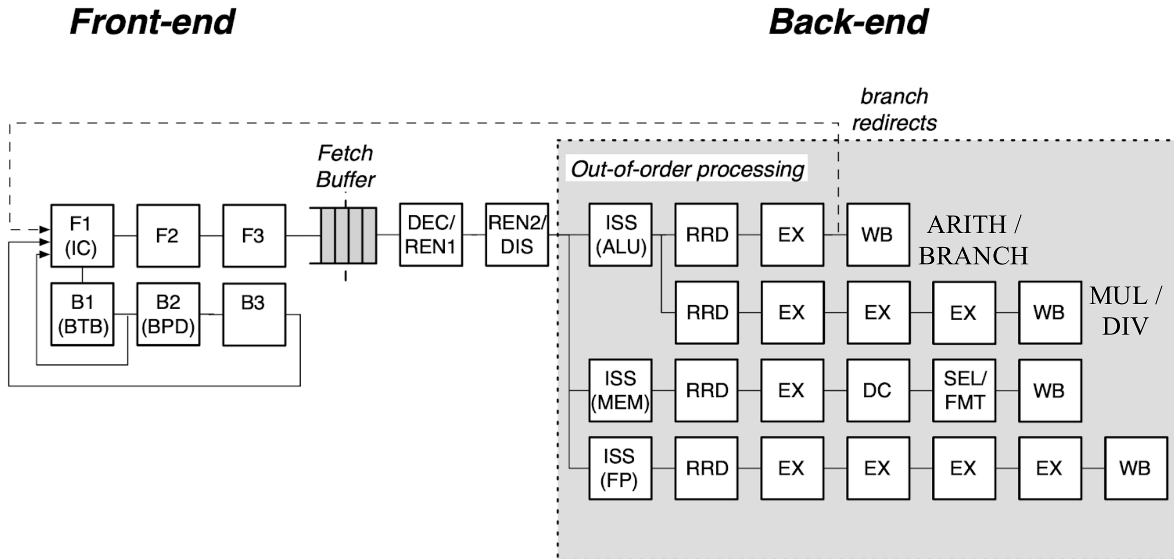
Label	ALU1	ALU2	Load	Store	FP
init:	beq a3, r0, done	addi t0, r0, 0			
loop:	addi a0, a0, 0x8	addi t0, t0, 0x1	fld f0, 0(a0)		
	addi a1, a1, 0x8		fld f1, 0(a1)		
					fadd.d f0, f0, f1
	bne t0, a3, loop	addi a2, a2, 0x8		fsd f0, 0(a2)	
done:	jr ra				

B) (12 points, 152 ONLY) Schedule the algorithm on the VLIW machine using software pipelining (you do not need to unroll the loop). Try to minimize the number of cycles, but prioritize correctness!

Label	ALU1	ALU2	Load	Store	FP
init:	beq a3, r0, done	addi t0, r0, 0			
	addi a1, a1, 0x8		fld f1, 0(a1)		
	addi a0, a0, 0x8		fld f0, 0(a0)		
					fadd.d f0, f0, f1
loop:	addi a1, a1, 0x8		fld f1, 0(a1)		
	addi a0, a0, 0x8		fld f0, 0(a0)		
	addi a2, a2, 0x8	addi t0, t0, 0x1		fsd f0, 0(a2)	
	bne t0, a3, loop				fadd.d f0, f0, f1
done:	jr ra				

Problem 3: Unified Physical Register File Out-of-Order Machines

Throughout this question, assume the following machine specifications:



- The machine can fetch, dispatch, issue, and commit at most **one** instruction per cycle.
- The processor runs the RISC-V instruction set with the F and D extensions.
- 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 window at the end of the REN2/DIS stage.
- Instructions are released from the issue window in the ISS stage.
- Commit is handled by a decoupled unit that looks at the ROB entries.
- Jump instructions issue and complete immediately on the same cycle that they dispatch.
- Assume all jump targets are perfectly predicted.
- Instructions may issue as soon as the same cycle that the writer of their last outstanding operand is in the writeback stage.
- **Ignore structural hazards on the register file ports**
- **Each functional unit has its own issue window, separate from the ROB**

Multiple Choice

(mark ALL that apply!)

A) **(2 points)** Which of the following fields are part of an issue window entry in **this** machine?
[X] Physical destination register
[ ] Architectural destination register
[ ] Last physical destination register
[X] Source present bits
[X] Operand physical register specifiers
[ ] Operand data
[ ] A flag to mark if the instruction has caused an exception

B) **(1 point)** An instruction in an issue window is guaranteed to also be in the ROB.
[X] True [ ] False

C) **(1 point)** An instruction in the ROB is guaranteed to also be in an issue window.
[ ] True [X] False

D) **(1 point)** An instruction enters the issue window in what phase of execution?
[X] Dispatch [ ] Issue [ ] Fetch

E) **(1 point) In a data-in-ROB design**, which of the following acts as a source for operands?
[X] Architectural register file
[ ] Physical register file
[X] ROB

F) (14 points) Consider the following code sequence that begins at address 0x00010000

```
loop:  fld    f0, 0(a0)
       fld    f1, 0(a1)
       fadd.d f0, f0, f1
       fsd    f0, 0(a2)
       addi   a0, a0, 0x8
       addi   a1, a1, 0x8
       addi   a2, a2, 0x8
       addi   t0, t0, 0x1
       bne    t0, a3, loop
```

Assume that the machine enters this loop with all instructions fetched, zero valid entries in the ROB, and the following initial rename table and free list contents before the first `fld` enters the ROB. Dequeue free list entries from the top.

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

Arch. register	Phys. register
a0	p1
a1	p5
a2	p33
t3	p17
t0	p41
f0	p62
f1	p28

Free List
p4
p55
p18
p30
p39
p11
p59
p60

Now consider the case in which the `fsd` takes an exception. Fill in the following table which describes the execution of each instruction) for eight instructions, beginning with the first fld. In the "Time" columns, fill in the cycles in which the instruction dispatches, issues, completes, and commits (if it commits), respectively.

Notes:

- The class was split about 50/50 between two cases. In one case, it was assumed that one instruction per issue queue could issue in a given cycle. In the other, the older `fadd.d` issues before the newer `addi`, blocking the `addi` for one cycle. Therefore, both answers were considered correct, and solutions are listed for both cases.
- You were free to choose any number as the start cycle, and the rest of the boxes were graded based on their value *relative to the start cycle*. I chose zero, but many of you chose 1, 2, or even 5!
- No points were deducted if PR1 and PR2 were swapped for the stores, but the order in the chart with the base address as PR1 is the canonical ordering.
- The cycle for the store to "complete" was somewhat tricky to interpret, so any value from 3-5 cycles after its issue was accepted. I used 3 cycles as the baseline, since that is when it goes to the data cache.
- The store does not actually commit!

Assuming the newer instruction waits for the older instruction:

PC	Physical Register Specifiers				Cycle #			
	PRd	LPRd	PR1	PR2	Dispatch	Issue	Complete	Commit
0x00010000	p4	p62	p1	--	0	1	6	7
0x00010004	p55	p28	p5	--	1	2	7	8
0x00010008	p18	p4	p4	p55	2	7	13	14
0x0001000C	--	--	p33	p18	3	13	16	--
0x00010010	p30	p1	p1	--	4	5	8	--
0x00010014	p39	p5	p5	--	5	6	9	--
0x00010018	p11	p33	p33	--	6	8	11	--
0x0001001C	p59	p41	p41	--	7	9	12	--

Assuming one instruction *per issue queue* can issue per cycle:

PC	Physical Register Specifiers				Cycle #			
	PRd	LPRd	PR1	PR2	Dispatch	Issue	Complete	Commit
0x00010000	p4	p62	p1	--	0	1	6	7
0x00010004	p55	p28	p5	--	1	2	7	8
0x00010008	p18	p4	p4	p55	2	7	13	14
0x0001000C	--	--	p33	p18	3	13	16	--
0x00010010	p30	p1	p1	--	4	5	8	--
0x00010014	p39	p5	p5	--	5	6	9	--
0x00010018	p11	p33	p33	--	6	7	10	--
0x0001001C	p59	p41	p41	--	7	8	11	--

Problem 4: Branch Prediction

A) **(1 Point)** Which of the following language-level constructs are typically compiled to use register indirect jumps? Check all that apply.

[X] Case statements
[X] Subroutine returns
[X] Dynamically dispatched function calls

B) **(1 Point)** How many bits of global history are required to perfectly predict the direction of the branch at label F? Check one answer.

C code	Assembly

<pre>if (a == 2)   // A
  b = a;   // B
if (b > c) { // C
  d = 0;   // D
} else {
  d = 1;   // E
}
if (d != 0)   // F
  e = 0;   // G
...   // H</pre>
	<pre>A: li t0, 0x2
	   beq a0, t0, C
	B: mv a1, a0
	C: blt a1, a2, E
	D: li a3, 0x0
	   j F
	E: li a3, 0x1
	F: beq a3, r0, H
	G: li a4, 0x0
	H: ...</pre>

[ ] 0 [X] 1 [ ] 2 [ ] 3

C) **(2 Points)** Why don't machines speculatively execute down both branch directions?

(See lecture 12, slide 14)
If a machine speculatively executes down both sides of the branch one of those paths is *guaranteed* to be a mis-speculation. This forces the machine to flush mis-speculated instructions on every branch and uses resources inefficiently. Conversely, since modern branch predictors are so accurate, machines can generally speculate down the correct path through multiple branches. This fills the pipeline with useful instructions, more efficiently using superscalar resources.

Longer discussion:
The case for speculating down both sides gets worse if the machine needs to speculate again before the branch is resolved, as will be the case with a deep superscalar pipeline. If we encounter another branch on both sides of the first branch and speculate down those, only ¼ of the explored is going to be correct. Moreover, mis-speculation is harder to recover from, as all younger instructions aren't necessarily on the mis-speculated path.

For the remainder of this problem, we'll consider the following code, which counts the number of false to true transitions in an array of C booleans.

These code listings are provided in appendix B.

```
bool array[N] = {…};
int posedge = 0;
for (int i = 1; i < N; i++) {
  if (array[i] && !array[i-1])
      posedge++;
}
```

Specifically, we'll consider the following assembly implementation of the loop above.

```
// a0 holds N
// a1 holds array
      li       a2, 0          // Initialize posedge
      add      a3, a1, a0     // Set up loop bound
loop:
      addi     a1, a1, 1      // Bump pointer
      bge      a1, a3, done   // Check loop condition
      lbu      a4,  0(a1)     // Load current element (array[i])
      lbu      a5, -1(a1)     // Load previous element (array[i-1])
      sltu     a4, x0, a4     // Set a4 to 1 if nonzero, else zero a4
      bgeu     a5, a4, loop   // Branch if not posedge (prev nonzero or equal)
      addi     a2, a2, 1      // Increment posedge
      j loop
done:
```

D) **(7 points) Branch History Table (BHT)**

The processor that this code runs on uses a 512-entry branch history table (BHT), indexed by PC [10:2]. Each entry in the BHT contains a 2-bit counter, initialized to the 10 state (weakly taken).

Each 2-bit counter works as follows: the state of the 2-bit counter decides whether the branch is predicted taken or not taken, as shown in the table below. If the branch is actually taken, the counter is incremented (e.g., state 00 becomes state 01). If the branch is not taken, the counter is decremented. The counter saturates at 00 and 11 (a not-taken branch while in the 00 state keeps the 2-bit counter in the 00 state)

State	Prediction
00	Not taken
01	Not taken
10	Taken
11	Taken

Assuming array = {0,1,0,1,0,1,0}, fill out the following tables. Each table corresponds to one branch and their respective BHT entries. Each row corresponds to one execution of the branch. Fill it out as follows:

- For the **Prediction** column: use **T** for Taken and **NT** for Not Taken
- For the **Correct** column: use **Y** to indicate a correct prediction and, **N** for incorrect
- For the **State** column: write the state of the entry on that cycle {**00, 01, 10, 11**}

Finally, fill out the total number of correct predictions in the boxes at the bottom of the table. The first two branches have been filled out for you.

bge (loop condition)		
State	Prediction (T / NT)	Correct? (Y / N)
10	T	N
01	NT	Y
00	NT	Y
00	NT	Y
00	NT	Y
00	NT	Y
00	NT	N
Total Correct:		5

bgeu (skip condition)		
State	Prediction (T / NT)	Correct? (Y / N)
10	T	N
01	NT	N
10	T	N
01	NT	N
10	T	N
01	NT	N
10		
Total Correct:		0

E) **(2 points)** Suppose we keep two bits of global branch history which we use to index into one of four BHTs each with the same structure as the BHT in part A. What sort of branch correlation can this predictor resolve that the BHT in part A cannot?

_______Spatial_____ Correlation

F) **(7 points)** Suppose we ran the code from part E on a long input array (N > 100000), and that the input array's values alternate every element (i.e. {0, 1, 0, 1…}). Give the final state of all entries of the predictor that could be indexed by the two branches. **If an entry is never indexed, leave it blank.** How accurate is this predictor over the entire execution of the loop? Explain briefly how you arrived at your solution.

Assume:

- The global history register is initially 01, with the LSB indicating the most recent branch
- All BHT entries are initially 10 (weakly taken), as in part E
- N is odd

BHT Entry	BHT 0	BHT 1	BHT 2	BHT 3
bge (loop)	00	01		
bgeu (skip)	11		00	

The trick here is to note that, since the skip condition is taken every other iteration, global history will be {00, 01} on the BGE and {00, 10} on the BGEU for {no posedge, posedge [on the last iteration]} respectively. Thus we need only consider those four entries. In steady state:

BGE is never taken → both entries should be 00
BGEU is *always* taken if GH = 00 (no posedge on last iteration) → BHT0 entry = 11
BGEU is *never* taken if GH = 10 (posedge in the last iteration) → BHT2 entry = 00

Thus after some training, all branches are predicted with **100% accuracy** -- our final answer. In steady state the table should look like:

BHT Entry	BHT 0	BHT 1	BHT 2	BHT 3
bge (loop)	00 (Strongly NT)	00 (Strongly NT)	?	?
bgeu (skip)	11 (Strongly T)	?	00 (Strongly T)	?

That's the bulk of the question. For full credit we need to consider the startup conditions, which may touch other entries, and loop exit, which will perturb the steady-state condition.

For startup:
Loop iter 1:
BGE 1: BHT 1 Predict T; Mispredict -> History: 01 -> 10; State: 10 -> 01 *
BGUE 1: BHT 2 -> Predict T; Mispredict -> History: 10 -> 00; State 10 -> 10 *
Loop iter 2:
BGE -> BHT 0 -> Predict T; Mispredict -> History 00 -> 00; State 10 -> 01*
BGUE -> BHT 0 -> Predict T; Correct -> History 00 -> 01; State 10 -> 11*
* : Now trained to predict in the correct direction

At this point we've hit the recurrence and all relevant BHT entries are adequately trained (but not at their final states). We've only used entries that will access in steady state → no changes to the steady state table above.

For exit:

Since N is odd, the final skip condition will be taken (no posedge) → global history before the branch out of the loop will thus be 01. On the final BGE we predict NT, so we increment that entry.

Prediction accuracy:	100%

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.

Notes:

- 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]`

Appendix B: Code Listings for Question 4

C implementation:

```
bool array[N] = {…};
int posedge = 0;
for (int i = 1; i < N; i++) {
  if (array[i] && !array[i-1])
      posedge++;
}
```

Assembly implementation under consideration.

```
// a0 holds N
// a1 holds array
      li       a2, 0          // Initialize posedge
      add      a3, a1, a0     // Set up loop bound
loop:
      addi     a1, a1, 1      // Bump pointer
      bge      a1, a3, done   // Check loop condition
      lbu      a4,  0(a1)     // Load current element (array[i])
      lbu      a5, -1(a1)     // Load previous element (array[i-1])
      sltu     a4, x0, a4     // Set a4 to 1 if nonzero, else zero a4
      bgeu     a5, a4, loop   // Branch if not posedge (prev nonzero or equal)
      addi     a2, a2, 1      // Increment posedge
      j loop
done:
```

Reference input: {0, 1, 0, 1, 0, 1, 0}

BHT entry state table and predictions.

State	Prediction
00	Not taken
01	Not taken
10	Taken
11	Taken