**CS152 Final Review (Part 1)**

**Q1. Iron Law**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Instructions/ program | CPI | Cycle time | Execution time |
| Pipelining a single-cycle implementation |  |  |  |  |
| Adding stages to an existing pipeline |  |  |  |  |
| Adding bypass paths |  |  |  |  |
| Adding hardware floating-point instructions |  |  |  |  |

**Q2. OoO Processor – Data-in-ROB design**

Consider the following Data-in-ROB OoO processor:

* ROB has 12 entries, from ROB0 to ROB11
* If the instruction producing a source register has committed before the dependent instruction enters the ROB, use the architectural register name
* Each cycle, two instructions can dispatch, and if multiple instructions can issue or complete, then order instructions are chosen
* Each cycle, one instruction of each type (ALU or LD/ST) can issue or complete
* Each cycle, two instructions can commit, and if multiple instructions can issue or complete, then order instructions are chosen.
* 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**, fully pipelined
* ALU instructions take **one cycle**, and branches resolve/complete using the ALU one cycle after they issue. ALU is fully pipelined.
* Branch is predicted always taken
* Assuming branch mispredictions are handled more quickly than exceptions

Now we will run a strcpy() program, which takes string “H” as the input (a string with only one character followed by a NUL terminator).

loop: lb t0, 0(a0)

sb t0, 0(a1)

addi a0, a0, 0x1

addi a1, a1, 0x1

bne t0, x0, loop

Fill the table out through when the mispredicted branch is caught.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | cycle # | | | | Data | | |
| Instruction | dispatch | issue | complete | commit | rd/ROB | rs1 | rs2 |
| lb t0,0(a0) | 0 | 1 | 4 | 5 | ROB0 | a0 | -- |
| sb t0,0(a1) | 0 |  |  |  |  |  |  |
| addi a0,a0,0x1 |  |  |  |  |  |  |  |
| addi a1,a1,0x1 |  |  |  |  |  |  |  |
| bne t0,x0,loop |  |  |  |  |  |  |  |
| lb t0,0(a0) |  |  |  |  |  |  |  |
| sb t0,0(a1) |  |  |  |  |  |  |  |
| addi a0,a0,0x1 |  |  |  |  |  |  |  |
| addi a1,a1,0x1 |  |  |  |  |  |  |  |
| bne t0,x0,loop |  |  |  |  |  |  |  |
| lb t0,0(a0) |  |  |  |  |  |  |  |
| sb t0,0(a1) |  |  |  |  |  |  |  |
| addi a0,a0,0x1 |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

1. What is the last commit instruction?
2. What limits the IPC for an infinitely long string?

**Q3. Branch Prediction – BTB**

Consider a fully bypassed 5-stage RISC-V processor. Assume we have added a 4-entry fully associative BTB to the Fetch Stage. The BTB is fully searched in the Fetch Stage to see if the PC matches any valid tags. If there’s a match, the BTB makes a prediction (i.e., redirects the PC to the target PC recorded in the BTB).

For a JAL instruction (J-type instructions), the 25-bit jump target offset is sign-extended and shifted left one bit to form a byte offset, then added to the pc to form the jump address. Note we have an adder to calculate the jump address for JAL instructions in the decode stage. JAL stores the address of the instruction following the jump (pc+4) into register x1.

For a JALR instruction (I-type instruction), the jump address is obtained by sign-extending the 12-bit immediate then adding it to the address contained in register rs1, and hence is only known in the execute stage. The address of the instruction following the jump (pc+4) is written to register rd. Note that the instruction “jr ra” is equivalent to “jalr x0,x1,0”.

0x2000: jal foo  
0x2004: addi x3,x0,3  
0x2008: jal foo  
0x200c: sub x4,x5,x7  
0x2010: lw x7,4(x6)  
…

foo: 0x4000: and x10,x11,x12  
0x4004: jalr x0,x1,0

bar: 0x4008: xor x20,x21,x22  
0x400c: or x24,x25,x26  
0x4010: jalr x0,x1,0

Initially, the BTB contains:

|  |  |  |
| --- | --- | --- |
| Tag | Valid | Target PC |
| 0x2000 | 1 | 0x4000 |
| 0x4004 | 1 | 0x2004 |
| 0x2008 | 0 | 0x4008 |
| 0x4010 | 0 | 0x2010 |

What is the final state of the BTB when the PC becomes 0x2010?

|  |  |  |
| --- | --- | --- |
| Tag | Valid | Target PC |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Q4. VLIW – 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).

1. 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)?
2. 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. 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?

**Q4. Vectors – Vectorization**

Vectorize the following double-precision dot product C code using the RISC-V vector ISA. 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;

}

1. 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.

1. If the ISA did not support variable-length vectors (i.e., no vsetvl), like packed SIMD, discuss two ways the code could be modified to support n not evenly divisible by MVL.