EECS 361

Midterm Solutions

-1 point for any incorrect calculation.

Q1) System Performance (20 points)

The base system spends 82% of the time computing and 18% of the time waiting for the disk. Integer instructions (40% of executed instructions) have a CPI of 1, floating-point instructions (30%) have a CPI of 5, and other instructions (30%) have a CPI of 2.

i. a) The processor is replaced with one that reduces computation time by 35%. (5 points)

Speedup = 1 / ((1 - 0.82) + 0.82 * 0.65) = 1.40

b) The disk is replaced with one that reduces disk wait time by 85%. (5 points)

Speedup = 1 / ((1 - 0.18) + 0.18 * 0.15) = 1.18

c) The floating-point CPI is changed to 3. (5 points)

Average CPI (old) = 0.40 * 1 + 0.30 * 5 + 0.30 * 2 = 2.5
Average CPI (enhanced) = 0.40 * 1 + 0.30 * 3 + 0.30 * 2 = 1.9

Speedup (computation) = 2.5 / 1.9 = 1.316

Speedup = 1 / ((1 - 0.82) + 0.82 / 1.316) = 1.245

ii. Part a results in the best speedup (1 point)

iii. If the disk was infinitely fast: (4 points)
speedup = 1 / (1 - 0.18) = 1.22 <= still slower than part a

If the floating-point computation was infinitely fast:
Average CPI (old) = 2.5 (from part i)
Average CPI (enhanced) = 0.40 * 1 + 0.30 * 0 + 0.30 * 2 = 1

Speedup (computation) = 2.5 / 1 = 2.5

Speedup = 1 / ((1 - 0.82) + 0.82 / 2.5) = 1.969 <= better speedup than part a

If the floating-point computation was close to infinitely fast (CPI = 1):
Average CPI (old) = 2.5 (from part i)
Average CPI (enhanced) = 0.40 * 1 + 0.30 * 1 + 0.30 * 2 = 1.3

Speedup (computation) = 2.5 / 1.3 = 1.923

Speedup = 1 / ((1 - 0.82) + 0.82 / 1.923) = 1 / 0.6064 = 1.649 <= better speedup than part a

If the floating-point computation was close to just faster (CPI = 2):
Average CPI (old) = 2.5 (from part i)
Average CPI (enhanced) = 0.40 * 1 + 0.30 * 2 + 0.30 * 2 = 1.6

Speedup (computation) = 2.5 / 1.6 = 1.5625

Speedup = 1 / ((1 - 0.82) + 0.82 / 1.5625) = 1 / 0.7048 = 1.4188 <= better speedup than part a

Wrong formula = 0 points.

Some people chose to combine the speedups for some reason. So the answer would be

1 / ((0.82*.65) + (0.18*.15)) =1 / (0.533 + 0.027) = 1 / 0.56 = 1.7857

Only 2 points for correct answer with not enough little work.

Q2) MIPS ISA (35 points)

1 point for trying on any of them for anything related.

i. In MIPS, why is the offset of a branch instruction from the PC of the next instruction instead of the PC of the current instruction? (8 points)

The PC of the next instruction can be computed in the first stage of the pipeline. By making the result of the branch be at this value + offset, only one more arithmetic operation is necessary in the event of a branch. If the branch target was the PC + offset instead of PC + 4 + offset, the current value of the PC would have to be subtracted by 4 before computing the branch target (or of course both PC + 4 and PC would have to be stored).

Since PC + 4 is computed anyway, 4 points.
Must explain that all we have is PC + 4 for full credit.

ii. Why are the offsets for branch instructions and displacements for load and store instructions in the MIPS ISA limited to 16-bits? (8 points)

The opcode and two register addresses use up 16 bits of the instruction. To support offsets greater than 16 bits, longer fixed-length instructions or variable-lengthed instructions would be required.

4 points for saying the 16 bits is enough to jump. 2 points for saying to keep instructions small.

iii. Why is the branch offset shifted left by 2 bits while the displacement for loads and stores are not shifted? (9 points)

4 points for branch offset.

Because instructions are four bytes long, their addresses always have their two least-significant bits = 0, which is what the shift accomplishes.

5 points for explaining load and stores.

Data needs to be byte-addressable, so displacements are not shifted.

Q3) Pipelining (25 points)

i. In a typical 5 or 6 stage pipeline, the CPI might be in the range of 1.0 to 1.5. Does this mean that most instructions have a latency of 1 or 2 cycles? (8 points)

No; in a pipeline, an instruction may be completed every cycle or two due to parallelism and overlap, but the instruction latency is still bound by the length of the pipeline. Throughput and latency are different.

Correct answer without explanation is 3 points. Incorrect explanation and correct answer is 0 points.

ii. Why do conditional branches impact the performance of a pipelined implementation? (8 points)

Conditional branches present a control hazard that can stall instruction fetch and thus create bubbles in the pipeline.

iii. 3 solutions to reduce impact of branches in a pipeline: (9 points)

- Delayed branches execution: change the semantics of the branch to always execute the instruction immediately following the branch regardless of branch outcome, and have the compiler insert a non-dependent instruction in this branch delay slot.

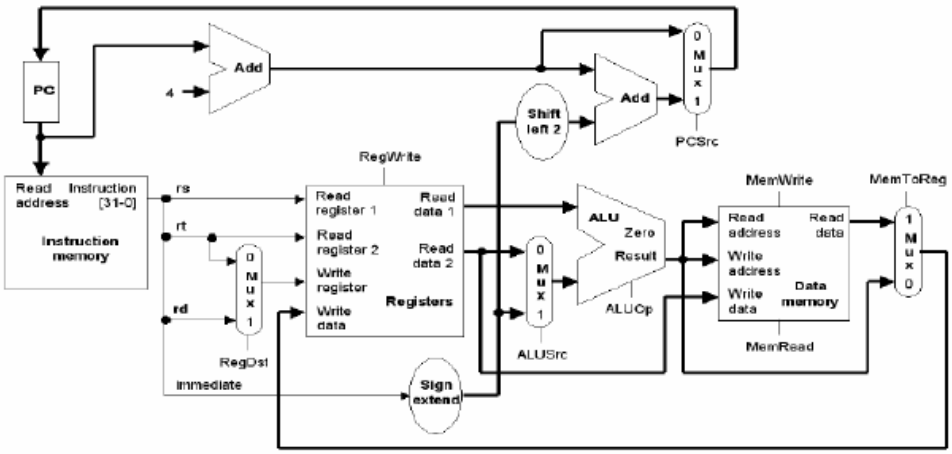
- Any kind of static or dynamic branch prediction: Continue executing instruction from the not-taken path of the branch and squash instructions if the branch is taken.

- Move up the branch resolution point: resolve branches in the ID stage rather than in the MEM or WB stages, for example using SET instructions following by simple branch instructions.

-Use redundant hardware to calculate both paths until the branch is resolved

1 point for data forwarding but not full credit since it's a general solution and assumed for pipelined.

Q4) Single-Cycle Datapath & Control (10 points)



(a) We wish to implement a new I-type instruction `sw addi` on the single-cycle datapath shown above. The instruction `sw addi $rs, $rt,` immediate (offset is the normal sign extended immediate value) has the following meaning :

```
sw $rt, offset($rs)
addi $rt, $rs, immediate
```

Show that the given datapath can support this new operation simply by setting the control signals appropriately. Fill in the missing values below. Use **X** to indicate "don't-care" values:

Control signal	Value
RegDst	0
RegWrite	1
ALUSrc	1
ALUOp	add
MemWrite	1
MemRead	X
MemToReg	0
PCSrc	0

Note: The key to this problem is to realize that sw addi is just like sw except that the value "rs + immediate" computed in the ALU needs to be stored in the rt register.

Anything wrong is -2 points. Not specifying the don't care is -1 point.

Q5) Caches (20 points)

1 point per part (tag, index, block offset, and byte offset). For fully associative, 2 points on index.

i) (16 points)

direct mapped: Since there are 4 words per block and 256K words in the cache, there are 256K/4 = 64K blocks in the cache. Thus, log(64K) = 16 bits of the address are needed to specify the cache block. The remaining bits are used for the tag. The partition would be:

bits 0-1: byte offset (2)
2-3: word offset (2)
4-19: cache block index (16)
20-31: tag (12)

2-way set associative: Since there two blocks per set, and 64K blocks in the cache (see above), there must be 32K sets. Thus, the number of bits required to specify a set is log(32K) = 15. The remaining bits are used for the tag, as follows:

bits 0-1: byte offset (2)
2-3: word offset (2)
4-18: set index (15)
19-31: tag (13)

4-way set associative: Since there four blocks per set, and 64K blocks in the cache (see above), there must be 16K sets. Thus, the number of bits required to specify a set is log(32K) = 14. The remaining bits are used for the tag, as follows:

bits 0-1: byte offset (2)
2-3: word offset (2)
4-17: set index (14)
18-31: tag (14)

fully associative: Since a block can be placed anywhere in the cache, all the remaining bits (other than the word and byte offsets) must be used as the tag. Hence,

bits 0-1: byte offset (2)
2-3: word offset (2)
4-31: tag (28)

ii. Cache misses can be characterized as one of the following: compulsory misses, capacity misses, and conflict misses. Describe how each of these kinds of misses can be addressed in the hardware. (4 points)

Compulsory misses, i.e. cache misses that occur the first time a datum or instruction is accessed, can be often reduced by increasing the size of the cache block. Since increasing the size of the cache block causes more neighboring locations to be cached when a cache miss occurs, it increases the likelyhood that a datum that has not yet been accessed will

be in the cache later on when it is accessed. (i.e. better data locality). Writing that increasing the size of the cache is too general, unless a good explanation is given.

UPDATE: Since the lecture slides and book refer to the compulsory miss as the first miss to a block (not a first miss to a location), explaining that nothing can be done is also an acceptable answer.

Capacity misses, i.e. cache misses that occur because the cache isn't large enough to hold all the blocks that a program is currently accessing, can be reduced by increasing the size of the cache.

Conflict misses, i.e. cache misses that occur because two or more blocks currently in use map to the same location in the cache, can be reduced by increasing the associativity of the cache. The greater the associativity, the larger the set size. In this way, two blocks that map to the same set can be accommodated within the set. Larger number of indexes works too.