Only Problem Set Two will be graded. Turn in only Problem Set Two on October 19, 2016 (Wednesday) by 5:00 pm.
Submit homework via gradescope <gradescope.com> as a PDF. Other formats may or may not be viewed and evaluated accurately.

Problem Set One

1. Hennessy & Patterson (5th Ed.) 1.1 (Chip Manufacturing)
2. Hennessy & Patterson (5th Ed.) 1.8 (Moore’s Law)
3. Hennessy & Patterson (5th Ed.) 1.14 (Speedups)
4. Hennessy & Patterson (5th Ed.) A.8 (Instruction Encoding)
5. Hennessy & Patterson (5th Ed.) A.18 (ISA Styles)
6. Hennessy & Patterson (5th Ed.) C.4 (Hazards of Pipelining)
7. Hennessy & Patterson (5th Ed.) C.5 (Addressing Modes)
8. Hennessy & Patterson (5th Ed.) C.6 (Register-memory Pipelining)
Problem Set Two

1. Amdahl’s Law

(15 points)

A small startup company with big dreams is designing a multi-core processor to compete with AMD. The company has hired you to help them determine the pricing of their processor. In performing your calculations, you can make the following assumptions:

• Die cost is inversely proportional to the number of good dies per wafer.
• The number of dies per wafer is inversely proportional to die area (note that this is slightly different from your book, which also includes a term for wasted dies in the wafer margins).
• The area of an $n$ core die is $n$ times the area of a single core die.
• The wafer yield is 100%, the cost per wafer is $800, and the wafer area is 1000 cm$^2$.

As you recall, the formula for die yield is:

$$\text{Die yield} = \text{Wafer yield} \times (1 + \text{Defects per unit area} \times \text{Die area})^{-N}$$

The engineers at the company inform you that the die yield for a die with only a single core is 80%, and that the cost of the die is $10. They tell you that the value for $N$ is 12.

(a) Using the above information, what is the expected defects per core?

(b) If the processor will have $n$ cores per die, what is the cost per die in terms of $n$?
(c) What is the area of each die?

(d) After analyzing the programs run by a typical user, the engineers determine that each time the number of cores double, the execution time of 40% of the typical program will be halved. The execution time of the remaining 60% of the typical program is unaffected by the number of cores. What is the expected overall increase in performance for an $n$ core system over a single core system? What is the asymptotic performance (i.e. if you had infinite cores)? What about the performance with 4 cores?

(e) One of the business analysts for the company is trying to find a function that relates the performance of the processor to the cost so that (s)he can more easily determine the optimal cost for the processor. Please help this business analyst determine a function which gives a good fit.
2. Delay Slot Machine

(30 points)

MIPS with its 5-stage pipeline requires a delay slot for branches because it can identify neither whether to take the branch nor where to jump to before the next instruction must be fetched. Instead of simply stalling on every branch, the designers of the ISA decided to allow compilers to put independent instructions after the branch into the branch’s ‘delay slot’. If the compiler cannot find an instruction that meets that constraint, it manually inserts a NOOP.

This works for MIPS with its single delay slot; however, with more complex ISAs, this delay grows longer, resulting in more delay slots.

You can consider the delay slots as partitioned as follows:

```
...  
A     
branch
m     
B     ...
C     ...
...  
```

A is the basic block preceding the branch. $m$ is the number of delay slots following the branch, $B$ is the block to execute if the branch is not taken, and $C$ if it is taken. Note that a basic block is a group of instructions without any control flow instructions within it.

(Part A) You initially explore computing both predicate and target at the same time, and end up with 5 delay slots on your branch. Since you have many more delay slots to fill, rather than allowing them to be filled only with instructions from A that are independent of the branch, you want compilers to be able to fill them with a mix of instructions from A, B, and/or C. Whichever instructions can be filled from A will be done first, the remaining delay slots being a mix of instructions from B and C.

(a) What constraints need to be adhered to in filling in delay slots? Please provide the least stringent set of constraints (without leading to semantic violations) so that you maximize delay slot fill-ins. (Pay attention to unintentional data hazards the compiler may introduce!)

To get you started, one constraint would be If two instructions from C are moved up and they have a data dependency, they must remain in the same order.
(b) According to your constraints, what is the largest set of instructions in the following example that can be moved into the delay slots? (You may want to provide a brief reasoning as to why particular instructions cannot be moved into the delay slots if you are unsure of full-credit.)

| A   | add R4, R4, 3  
|     | add R1, R2, R3  
|     | sub R2, R2, R3  |
| br  | bz R2, C  
|     | delay slots  |
| B   | mult R3, R3, R4  
|     | ld R2, [R5]  
|     | add R4, R5, R5  
|     | ...  |
| C   | add R4, R4, 8  
|     | lui R2, 15  
|     | div R2, R2, 2  
|     | ...  |

(Part B) In order to help solve this, you work some pipeline magic to evaluate the predicate of a branch (i.e. whether the branch will be taken) before evaluating the target address to go to if the branch is taken (C’s address). With this new pipeline, you know after 2 delay slots whether you will continue to block B or you must jump somewhere else. **If you are jumping to C however, you still need 3 more delay slots** (computing the target is hard when you have fancy addressing modes!). This new partition of the delay slots into m slots before the predicate is evaluated and n slots after, but before the target is evaluated, can be visualized as:

```
...  
A  
branch  
m  
n  
B  
...  
C  
...  
```

You decide to leave the semantics to the first 2 delay slots as you prescribed in Part A. However, through industrial espionage, you find a competing company is doing the same, and they have somehow enabled the compiler to fill those 3 delay slots much more effectively with fewer constraints to check!

Your espionage team has very few details. They know it involves a new adder; in parallel with the normal PC increment adder, this adder does a (+n), and then results from both adders go into a MUX. They also tell you that, somehow, for these n delay slots, dependencies between instructions from block B and from block C do not need to be checked!
(a) You are under intense pressure to figure out what they did so you can stay competitive. What is the solution?

(b) According to your solution, eliminate the NOOPs in the following code.

A

| add R4, R4, 3 |
| add R1, R2, R3 |
| sub R2, R2, R3 |

B

| mult R3, R3, R4 |
| ld R2, [R5] |
| add R4, R5, R5 |
| ... |

C

| add R4, R4, 8 |
| lui R2, 15 |
| div R2, R2, 2 |
| ... |
3. Pipelined Stack ISA

(55 points)

In this question, we will be exploring various issues surrounding the design of a pipeline. We will be taking a look at what changes need to be made to the MIPS pipeline from your book in order to adapt it to stack ISA.

In Table 1 we have given you a Stack ISA that we have tried to make lean and mean. Unless an operand is denoted in the instruction column, the operands shown in the second column will be found on the stack with the first at the top of the stack, and the second operand right below the first.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>op1, op2</td>
<td>PUSH(POP(op1) + POP(op2))</td>
</tr>
<tr>
<td>sub</td>
<td>op1, op2</td>
<td>PUSH(POP(op1) - POP(op2))</td>
</tr>
<tr>
<td>mult</td>
<td>op1, op2</td>
<td>PUSH(POP(op1) * POP(op2))</td>
</tr>
<tr>
<td>push X</td>
<td>offset (optional)</td>
<td>PUSH(X + POP(offset))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X is immediate or displacement</td>
</tr>
<tr>
<td>pop Y</td>
<td>offset (optional)</td>
<td>Mem[Y+POP(offset)] = POP(Top-Of-Stack)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y is a memory address or displacement</td>
</tr>
<tr>
<td>be</td>
<td>pred, address</td>
<td>if(POP(pred) == 0), goto PC + POP(address)</td>
</tr>
</tbody>
</table>

Table 1: Table of Operations

BE pops a condition and an address off the top of the stack. The branch is taken only if the condition is 1 (true), and branches to the PC-relative address that was popped off the top of the stack (+1 = next instruction, -4 = 4 instructions back).

POP and PUSH both have two addressing modes, which requires an extra bit to specify which form of the instruction is in use. To indicate which form of PUSH you are using, you can use PUSH.I (immediate) and PUSH.D (displacement) to make it clear which addressing mode you are using. Likewise, with POP you can use POP.D (displacement) or POP.M (memory) to indicate which addressing mode is being used.

The stages of the pipelined version of our Stack ISA are similar to those in the 5 stage MIPS pipeline from your book (shown in Figure 1); the IF stage fetches instructions, ID decodes the instruction and obtains operand values, the EX stage performs (along with a few other tasks) arithmetic operations in addition to calculating the effective address for the displacement addressing mode, the MEM stage accesses memory, and the WB stage writes the results back to the register file. However, in a Stack ISA there is a stack instead of a register file, with instructions extracting or placing data on it as needed. The processor can extract up to two operands off the top of the stack during the ID stage. When data is popped off the top of the stack, that data is not overwritten until a new piece of data is pushed to that same location.
(Part A) In the MIPS pipeline, the register file was used to decode operands. However, in a stack ISA the register file is replaced by the stack and all operands must now be taken from the stack. Consequently, there is now certain additional information that we must keep track of. In particular, the state of the stack pointer must be preserved and passed between instructions; as the stack pointer is also on the critical path of the processor, its value must also be updated in a timely manner - the updated value of the stack pointer must be available at the start of the ID stage so that instruction operands can be fetched.

(a) How should the ID stage be constructed to support the stack architecture? What forwarding paths are needed for the stack pointer? What new information needs to be passed between the pipeline registers?
(b) In order to handle branches, the stack ISA takes the top of the stack and the address operands during the ID stage. It then compares the value from the top of the stack to 0 to decide whether or not the branch should be taken by the end of the ID stage. What changes, if any, must be made to the pipeline in order to support branches?

(c) Assuming there is no branch prediction, would simply stalling when a branch is encountered work with a stack ISA? If so, how many cycles should the processor stall for when a branch is encountered? If not, why not?

(d) There are some differing opinions in your class on whether or not the processor requires space to store the stack. One group argues that due to the way instructions consume the stack, forwarding data between pipeline stages eliminates the need for any storage space for the stack. Another group claims that while forwarding data between pipeline stages means there will be fewer instructions that actually need to push data onto the stack in the WB stage, the processor still needs stack space to execute code that pushes data to the stack. Which of these groups is correct? Be sure to support your answer with examples.
(Part B) The **PUSH** and **POP** instructions in the stack ISA have displacement addressing modes. In MIPS, the displacement addressing mode requires calculating an effective address for a memory location. What changes, if any, are needed to support this in a stack ISA?

---

(Part C) In order to support the **ALU** operations in a stack ISA (ADD, SUB, and MULT), what changes, if any, must be made to the pipeline?

---

(Part D) As you will have noticed by now, each operation pushes new data to the stack and then the next operation will consume it on the next cycle (with a few exceptions). In order to avoid stalling and only completing one instruction at a time, we must forward data between stages.

(a) If there is a way to forward data between stages, let us know how it should be done. Also, please mention what stages we must forward data from. (Please optimize for performance, not cost.)

---

(b) What instructions, if any, need to receive their forwarded data at the **ID** stage? For each instruction that needs to receive data at the ID stage, please briefly mention why.

---

(c) Which instructions, if any, will require stalling through interlock to wait for data? For each instruction that requires stalling, please briefly explain why it will require stalling.
(Part E) Now that we have the basics of the pipeline under control it is time to be more aggressive and see if we can squeeze more performance out of it. As you remember, there exist a number of techniques that ensure that the CPI hit of simply stalling on a branch can be ameliorated.

(a) One of these techniques puts the onus for speeding up the performance of branches on the compiler writers by asking them to fill in the branch delay slots with useful instructions. As you remember, branch delay slots can be filled in from multiple locations in the code with some of these locations imposing more stringent requirements to make their selection feasible.

Can you remind us where the possible locations for branch delay slot filling and what the respective constraints that need to be fulfilled are? (As you can imagine, as stack architectures are a bit primitive, canceling instructions are not part of this ISA.)

(b) Now that we know which locations are possible to select branch delay slots candidates, we want to make your life easy by allowing you to select the location that is the easiest. Nonetheless, as you can imagine, filling such instructions has never been easy and perhaps even more so in stack architectures. As you ponder this question, you yearn for the simplicity of a register architecture where registers can save values that will not be used for awhile. In your darkest moments you think that this is hopeless and that branch delay slots can never be filled in while every once in a while a flicker of hope appears, which tells you that at least some instructions can be found, but in your utter confusion you can’t quite clearly describe which ones they are. When you give up for the night and decide to go for a nightcap, everything looks much more rosy and clear, and you see clearly that many instructions (perhaps as many as in a register ISA) can be used to fill the branch delay slot in this stack ISA pipeline. Now that you have put all this behind and with the clarity of mind that ensues in tackling your homework, which one of these three options do you think holds and why? If you vote in favor of partially or fully utilizing instructions to fill in the branch delay slot, which ones are they and why do they qualify? If you think it is hopeless, please provide a reasoning.
(c) The beauty of branch delay slot instructions is that they will be executed independent of the resolution of the branch. Of course, that puts the onus on the compiler writers and in these early years of computer science when stack ISAs were common, great compiler writers were a precious resource rarely encountered. Taking heed of the situation you decide to do a primitive form of branch prediction wherein you always predict not taken with the instructions that are in the prediction window cancelable upon receiving word that the branch should have been taken instead. Your first thoughts go to the question of the CPI impact on this. Some of your colleagues claim that the CPI will be no worse than a perfect branch delay slot fill strategy and that you are a genius who has improved CPI without relying on compiler writers. Some others claim that you could do even better and certainly no worse, particularly if you do good instruction scheduling and branch structuring (i.e. appropriately structuring which basic block is the taken and which the not taken). Another group claims that you could end up doing worse despite your best efforts compared to a well executed branch delay slot strategy. Which one(s) of these groups do you agree with and why?

(d) Now that we have the basics out of the way, one needs to also pay some attention to the implementation of this primitive and somewhat rigid branch prediction strategy. One of the stumbling blocks in implementation is what happens when the prediction is wrong. One of the issues is the stubbing out of the effects of the mispredicted instruction(s) but perhaps that is straightforward and not that much different than what we did in the basic 5 stage pipeline. Yet what about the stack pointer? As you remember from the early part of this question that you tackled, stack pointers need to be handled in a careful manner in this architecture. What changes, if any, do you propose to do in pipeline design to ensure that the stack pointer is not mishandled as a result of a mispredicted branch?