CS570 Advance Computer Architecture

Homework 8

1. NVM basic

a. What is NVM and PCM?

Non-volatile memory (NVM)

- NVM is a type of computer memory that has the capability to hold saved data even if the power is turned off. Unlike volatile memory, NVM does not require its memory data to be periodically refreshed. It is commonly used for secondary storage or long-term consistent storage.
- It is widely used in memory chips for USB memory sticks and digital cameras.
- Examples of NVM include: All types of read-only memory, hard disks, Optical disks.

Phase-change memory (PCM)

- Phase Change Memory is type of non-volatile computer memory (NVRAM) that stores data by altering the state of the matter from which the device is fabricated, meaning it changes back and forth between amorphous and crystalline states on a microscopic level. PCM is considered an emerging technology.
- Phase change memory technology is ideal for both non-volatile dual in-line memory modules (NVDIMMs) and non-volatile memory express (NVMe) solid state drives (SSDs), because it is much closer in speed to dynamic RAM (DRAM),
- PCM is also called "perfect RAM" (PRAM) because data can be overwritten without having to erase it first.

b. What are the issues of DRAM and NVM hybrid memory systems?

DRAM

- Dynamic random access memory (DRAM) is a type of semiconductor memory that is typically used for the data or program code needed by a computer processor to function.
- Issues in DRAM are Memory is volatile, Data in storage cells needs to be refreshed and It is slower than SRAM.
- DRAM latency problem has issues like Inefficient bulk data movement, DRAM refresh interference (while DRAM is being refreshed it can't all be accessed) and Cell latency variation, due to manufacturing variability.

NVM hybrid memory systems

- A hybrid memory system consists of DRAM and NVM, to reduce overall energy of the memory system.
- Hybrid memory systems pose a new challenge to on-chip cache management due to the asymmetrical penalty of memory access to DRAM and NVM in case of cache misses

c. What is the basic idea of data placement between DRAM and PCM?

A hybrid memory design, where NVM (e.g., PCM) and DRAM coexist in a compute node, NVM might have a particular address range assigned to it or it might act as a temporary buffer for the primary memory. In such a scenario, to gain the full advantage of the memory subsystem, a programmer or compiler would need to be explicit about the initial placement of the data on different types of memories

2. PIM basic

a. What is the motivation of Processing in Memory (PIM)?

Processing in memory is the integration of a processor with RAM (random access memory) on a single chip. The result is sometimes known as a *PIM chip*.

The movtivation behind PIM is that the processor has spent an increasing amount of time waiting for data to be fetched from memory. In effect, a processor is limited to the rate of transfer at the bottleneck. Processing in memory is one approach to overcoming the von Neumann bottleneck, which is a limitation on throughput caused by the latency inherent in the standard computer architecture.

b. Please give a successful example of PIM.

- The Data-Intensive Architecture (DIVA) system employs Processing-In-Memory (PIM) chips as smart-memory coprocessors. The DIVA project has built a prototype development system using PIM chips in place of standard DRAMs.
- Gather/Scatter DRAM

c. What are the barriers of adoption PIM?

Adopting this PIM new technology would need a new programming interface, compiler support + HW interfaces. A lack of OS level tools like schedulers, data mapping, ACLs to accurately exploit its inherent nature.

d. What is the relation between PIM and LPM?

- PIM can improve locality which reduces data movement, improve data access concurrency.
- LPM proposes to reduce overall data movement, improve data supply to the different layers and to mask the difference between layers.

e. Assume we have developed technical solutions for the barriers, what are the limitation and potential of PIM? Can PIM replace CPU or GPU in the future? Please provide your explanation.

It can't immediately replace a CPU or a GPU and can work soley as an accelerator. There is still no existing toolchain to use these tools effectively.

3. Please list as least 4 methods to solve data dependences in instruction processing pipeline.

A data dependency occurs when an instruction depends on the results of a previous instruction. A particular instruction might need data in a register which has not yet been stored since that is the job of a preceeding instruction which has not yet reached that step in the pipeline.

Methods to solve data dependences

- ***Instructions Reordering***: Re-ordering technique is very simple and efficient way to remove some data dependency that occur during execution instruction in pipelined. In many cases occur where reordering may affect the execution because some time order of the execution matter.
  Data Hazard Example: I : 8 R7 R7 II : R7 8 R3 III : R4 R5 – R6
  Re-ordered Instructions: I : 8 R7 R7 II : R4 R5 – R6 III : R7 8 R3
- ***Data Forwarding or bypassing***: Data forwarding also known as bypassing is an efficient way to solve data dependency in pipelined instruction execution. In the data forwarding or bypassing technique, normal processor is updated with special hardware. This technique the result of one stage is forward before complete execution of particular instruction. Where the result is require it is directly pass to that pipeline stage
- ***Pipeline Stalls*** : Data dependency can also be removed by inserting stalls or alter the normal flow of execution. Stalls are inserted to skip one stall cycle and instruction is waiting until other same instruction or depended instruction complete or data hazard is chance is leave. The simplest way to fix the hazard is to stall the pipeline. Stalling involves blocking flow of instructions until the result is ready to be used.
- ***Compiler based Scheduling***: Compiler based scheduling is also a technique that is used to remove hazard efficiently. Compiler first detects the Hazards then analyzes the instruction where dependencies, NOP or bubbles are inserted by the compiler between these instructions that has dependencies.
- ***Register Renaming:*** Pipelined issue of data dependency can be solved through register renaming that is used to remove false data dependencies b/w instructions in running state. Register operands of instructions are renamed. Register renaming can reduce the impact of WAR and WAW dependencies. WAR and WAW both are data hazards.

4. Please list as least 3 methods to solve control dependences in instruction processing pipeline.

This type of dependency occurs during the transfer of control instructions such as BRANCH, CALL, JMP, etc. On many instruction architectures, the processor will not know the target address of these instructions when it needs to insert the new instruction into the pipeline. Due to this, unwanted instructions are fed to the pipeline.

Methods to solve control dependences

- ***Pipeline stall cycles:*** Freeze the pipeline until the branch outcome and target are known, then proceed with fetch. Thus, every branch instruction incurs a penalty equal to the number of stall cycles. This solution is unsatisfactory if the instruction mix contains many branch instructions, and/or the pipeline is very deep.
- ***Branch delay slots:*** The ISA is constructed such that one or more instructions sequentially following a conditional branch instruction are executed whether or not the branch is taken. The compiler or

assembly language writer must fill these *branch delay slots* with useful instructions or NOPs (no-operation opcodes). This solution doesn't extend well to deeper pipelines, and becomes architectural baggage that the ISA must carry into future implementations.

- ***Branch prediction:*** The outcome and target of conditional branches are predicted using some heuristic. Instructions are speculatively fetched and executed down the predicted path, but results are not written back to the register file until the branch is executed and the prediction is verified. When a branch is predicted, the processor enters a *speculative mode* in which results are written to another register file that mirrors the architected register file. Another pipeline stage called the *commit* stage is introduced to handle writing verified speculatively obtained results back into the "real" register file. Branch predictors can't be 100% accurate, so there is still a penalty for branches that is based on the branch misprediction rate.
- ***Indirect branch prediction***: Branches such as virtual method calls, computed gotos and jumps through tables of pointers can be predicted using various techniques.
- ***Return address stack (RAS):*** Procedure returns are a form of indirect jump that can be perfectly predicted with a stack as long as the call depth doesn't exceed the stack depth. Return addresses are pushed onto the stack at a call and popped off at a return.

5. What are the conditions to make an ideal pipeline? Why instruction processing pipeline is not an ideal one?

Pipelining is the ability to overlap execution of different instructions at the same time. It exploits parallelism among instructions and is ***NOT*** visible to the programmer. *An ideal pipeline is one where each step does a little job of processing the instruction and where every step operates in parallel.*

Instruction processing pipeline

- An instruction pipeline increases the performance of a processor by overlapping the processing of several different instructions. Often, this is done by dividing the instruction execution process into several stages.
- An instruction pipeline often consists of five stages, as follows:
  a. Instruction fetch (IF). Retrieval of instructions from cache (or main memory).
  b. Instruction decoding (ID). Identification of the operation to be performed.
  c. Operand fetch (OF). Decoding and retrieval of any required operands.
  d. Execution (EX). Performing the operation on the operands.
  e. Write-back (WB). Updating the destination operands.

Instruction level pipeline isn't ideal as it presents structural stalls, data hazard stalls and control stalls

6. Solve 3.1 of the text (6th edition)

3.1 [10] <3.1, 3.2> What is the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.47 if no new instruction's execution could be initiated until the previous instruction's execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one-cycle branch delay slot.

Latencies beyond single cycle	
Memory LD	+3
Memory SD	+1
Integer ADD, SUB	+0
Branches	+1
fadd.d	+2
fmul.d	+4
fdiv.d	+10

```
Loop:   fld     f2,0(Rx)
I0:     fmul.d  f2,f0,f2
I1:     fdiv.d  f8,f2,f0
I2:     fld     f4,0(Ry)
I3:     fadd.d  f4,f0,f4
I4:     fadd.d  f10,f8,f2
I5:     fsd     f4,0(Ry)
I6:     addi    Rx,Rx,8
I7:     addi    Ry,Ry,8
I8:     sub     x20,x4,Rx
I9:     bnz     x20,Loop
```

Figure 3.47 Code and latencies for Exercises 3.1 through 3.6.

Instruction		CLK Execution	CLK Completion
FLD	F2,0(Rx)	1	4 (1+3)
FMUL.D	F2,F0,F2	5	9 (4+1+4)
FDIV.D	F8,F2,F0	10	20(9+1+10)
FLD	F4,0(Ry)	21	24(20+1+3)
FADD.D	F4,F0,F4	25	27(24+1+2)
FADD.D	F10,F8,F2	28	30(27+1+2)
FSD	F4,0(Ry)	31	32(30+1+1)
ADDI	Rx,Rx,#8	33	33 (32+1)
ADDI	Ry,Ry,#8	34	34 (33+1)
SUB	X20,X4,Rx	35	35(34+1)
BNZ	X20,LOOP	36	37 (35+1+1)

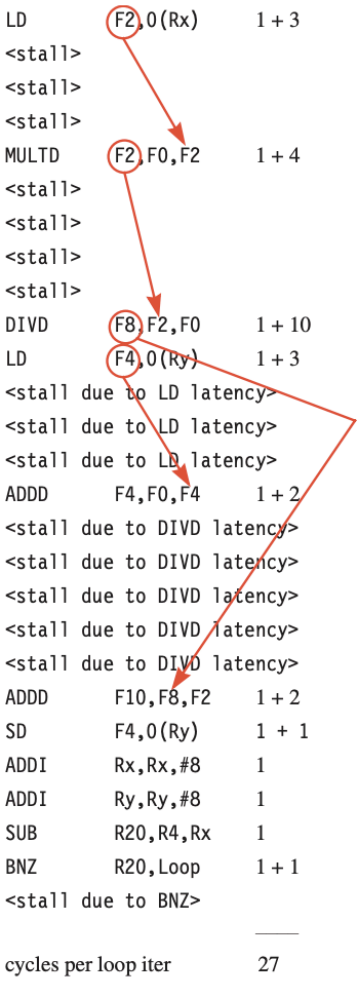
It takes 37 cycles to complete an iteration.

7. Solve 3.2 of the text (6th edition)

[10] <3.1, 3.2> Think about what latency numbers really mean—they indicate the number of cycles a given function requires to produce its output. If the overall pipe- line stalls for the latency cycles of each functional unit, then you are at least guar- anteed that any pair of back-to-back instructions (a "producer" followed by a "consumer") will execute correctly. But not all instruction pairs have a pro- ducer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 3.47 require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? Show the code with <stall> inserted where necessary to accom- modate stated latencies. (Hint: an instruction with latency +2 requires two <stall> cycles to be inserted into the code sequence.) Think of it this way: a one-cycle instruction has latency 1 + 0, meaning zero extra wait states. So, latency 1+1 implies one stall cycle; latency 1+N has N extra stall cycles.

SOLUTION:

Remember, the point of the extra latency cycles is to allow an instruction to complete whatever actions it needs, in order to produce its correct output. Until that output is ready, no dependent instructions can be executed. So the first LD must stall the next instruction for three clock cycles. The MULTD produces a result for its successor, and therefore must stall 4 more clocks, and so on.



```
LD        F2,0(Rx)      1 + 3
<stall>
<stall>
<stall>
MULTD     F2,F0,F2      1 + 4
<stall>
<stall>
<stall>
<stall>
DIVD      F8,F2,F0      1 + 10
LD        F4,0(Ry)      1 + 3
<stall due to LD latency>
<stall due to LD latency>
<stall due to LD latency>
ADDD      F4,F0,F4      1 + 2
<stall due to DIVD latency>
<stall due to DIVD latency>
<stall due to DIVD latency>
<stall due to DIVD latency>
<stall due to DIVD latency>
ADDD      F10,F8,F2     1 + 2
SD        F4,0(Ry)      1 + 1
ADDI      Rx,Rx,#8      1
ADDI      Ry,Ry,#8      1
SUB       R20,R4,Rx     1
BNZ       R20,Loop      1 + 1
<stall due to BNZ>

cycles per loop iter    27
```

It takes 27 cycles per loop iteration