Bitte bearbeiten Sie die folgenden Aufgaben in Zweiergruppen. Wenn Sie Feedback zu Ihren Lösungen haben möchten, können Sie diese in elektronischer Form per Mail an `h.g.esser@gmx.de` abgeben (bitte keine Mehrfachabgabe identischer Lösungen). Es gibt keine Bewertung/Benotung.

1. Bedingte Sprünge

a) Ein Programm bestehe zu 20 % aus bedingten Sprüngen, der Prozessor geht beim Befüllen der Pipeline davon aus, dass standardmäßig *nicht* gesprungen wird. Kommt es dann doch zum Sprung, müssen die zwei bereits in die Pipeline eingetragenen Befehle nach dem Sprung (einer in der F-Phase, einer in der D-Phase) gelöscht werden, die Pipeline wird neu gefüllt.

Zur Vereinfachung nehmen wir an, dass alle Befehle gleich schnell durch die Pipeline laufen. Wenn im Programm niemals gesprungen wird, dauert die Ausführung t Zeiteinheiten.

Nehmen Sie nun an, dass

(1) in 100 % aller Sprungbefehle doch gesprungen wird

(2) in 50 % aller Sprungbefehle doch gesprungen wird

(3) in 10 % aller Sprungbefehle doch gesprungen wird

und berechnen Sie für diese drei Fälle, wie stark sich die Ausführzeit durch das nun nötige Löschen der Pipeline verlängert. (Die Ergebnisse sollten jeweils die Form $k \cdot t$ haben, z. B. 1,2 t oder 1,7 t.)

b) Die MMIX-Plattform bietet Branch-Befehle auch in der Variante „probable branch“ (wahrscheinlicher Sprung) an. Erklären Sie, warum ein Programmierer durch richtigen Einsatz von Probable- und normalen Branch-Befehlen die Programmausführung beschleunigen kann.

2. Interrupts und Pipelines

Auf der Rückseite finden Sie einen Auszug aus Hennessy/Patterson: „Computer Architecture – A quantative Approach“. Lesen Sie den Auszug und beantworten Sie die folgenden Fragen.

a) Welche Rolle spielt der im Text erwähnte „exception status vector“?

b) Warum will man Exceptions in der richtigen Reihenfolge verarbeiten?

Auszug aus Hennessy/Patterson: „Computer Architecture – A quantative Approach“, 3rd ed., 2003, S. A-44 - A-45

Exceptions in MIPS

Figure A.28 shows the MIPS pipeline stages and which "problem" exceptions might occur in each stage. With pipelining, multiple exceptions may occur in the same clock cycle because there are multiple instructions in execution. For example, consider this instruction sequence:

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

Bezeichnungen hier:
IF = Instruction Fetch (F)
ID = Instruction Decode (D)
EX = Execute (X)
MEM = Memory Access (M)
WB = Write Back (B)

LD ist (irgend)ein Load-Befehl, DADD eine Integer-Addition von zwei Registern.

This pair of instructions can cause a data page fault and an arithmetic exception at the same time, since the LD is in the MEM stage while the DADD is in the EX stage. This case can be handled by dealing with only the data page fault and then restarting the execution. The second exception will reoccur (but not the first, if the software is correct), and when the second exception occurs, it can be handled independently.

In reality, the situation is not as straightforward as this simple example. Exceptions may occur out of order; that is, an instruction may cause an exception before an earlier instruction causes one. Consider again the above sequence of instructions, LD followed by DADD. The LD can get a data page fault, seen when the instruction is in MEM, and the DADD can get an instruction page fault, seen when the DADD instruction is in IF. The instruction page fault will actually occur first, even though it is caused by a later instruction!

Since we are implementing precise exceptions, the pipeline is required to handle the exception caused by the LD instruction first. To explain how this works, let's call the instruction in the position of the LD instruction i, and the instruction in the position of the DADD instruction $i + 1$. The pipeline cannot simply handle an exception when it occurs in time, since that will lead to exceptions occurring out of the unpipelined order. Instead, the hardware posts all exceptions caused by a given instruction in a status vector associated with that instruction. The exception status vector is carried along as the instruction goes down the pipeline. Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off (this includes both register writes and memory writes). Because a store can cause an exception during MEM, the hardware must be prepared to prevent the store from completing if it raises an exception.

precise exceptions: Pipeline kann bei Exception so angehalten werden, dass a) Befehle vor dem fehlerhaften Befehl noch komplett abgearbeitet werden, b) der fehlerhafte Befehl und alles dahinter komplett neu gestartet wird.

When an instruction enters WB (or is about to leave MEM), the exception status vector is checked. If any exceptions are posted, they are handled in the order in which they would occur in time on an unpipelined processor—the exception corresponding to the earliest instruction (and usually the earliest pipe stage for that instruction) is handled first. This guarantees that all exceptions will be seen on instruction i before any are seen on $i + 1$. Of course, any action taken in earlier pipe stages on behalf of instruction i may be invalid, but since writes to the register file and memory were disabled, no state could have been changed.

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

Figure A.28 Exceptions that may occur in the MIPS pipeline. Exceptions raised from instruction or data memory access account for six out of eight cases.