SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Lucas Biehl

Systemarchitektur SS 2021

Lösungsskizze 9

Aufgabe 9.1: Pipelining (1)

Verletzte Annahmen:

Homogene Teiloperationen: Gewisse Teiloperationen können nicht oder nur mit sehr viel Aufwand weiter aufgeteilt werden, beispielsweise Speicherzugriffe oder komplexe arithmetische Operationen. Beispielsweise müssten getaktete Register *innerhalb* des Speichers/ der arithmetischen Schaltung eingefügt werden.

Bei gewissen Speicherarten gibt es auch physikalische Prozesse, die nicht weiter geteilt oder parallelisiert werden können (beispielsweise die Bewegung eines Festplattenkopfes oder das Lesen eines Bits, also das Auslesen der Ladung eines Kondensators, im DRAM).

Damit sind einige Teiloperationen nach unten in ihrer Latenz beschränkt, so dass für eine vielstufige Pipeline die anderen Operationen weiter zerteilt werden müssen, womit die Latenz der verschiedenen Schritte nicht mehr homogen ist.

Wiederholung unabhängiger Operationen: Sowieso schon verletzt und nur durch Hazard-Unit oder Softwarebedingungen lösbar (unter Verlust von Teilen des idealisierten Speedups). Bei 100 Stufen gibt es *noch* mehr Möglichkeiten für logisch voneinander abhängige Operationen.

Aufgabe 9.2: Pipelining (2)

Die Daten- und Kontrollabhängigkeiten sind in Abbildung 4 dargestellt.

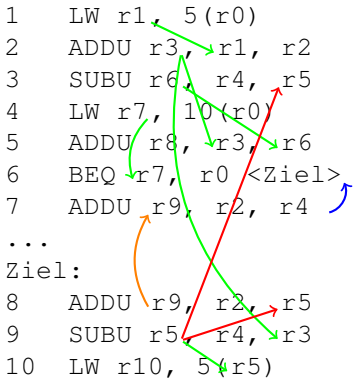

Abbildung 1: Daten- und Kontrollabhängigkeiten: RAW, WAW, WAR, und Control

Im Folgenden nehmen wir an, dass das General Purpose Register File bei fallenden Taktflanken schreibt.

Ohne Zusatzhardware: Durch das Pipelining können nur RAW- und Kontrollabhängigkeiten verletzt werden, da die Reihenfolge von schreibenden Zugriffen untereinander nicht verändert wird.

Folgende Punkte sind hier zu beachten:

- Nach einem Verzweigungsbefehl werden die nächsten 3 Takte möglicherweise nicht für relevante Rechnungen genutzt (wegen Ablaufstalling oder Branch Prediction).
- Zwischen einer Instruktion die in Register `rx` schreibt und einer nachfolgenden Instruktion, die aus `rx` liest, müssen mindestens 2 Instruktionen liegen.

Damit können wir folgende Ursachen für Hazards identifizieren:

- RAW Abhängigkeit zwischen 1 und 2 auf `r1`
- RAW Abhängigkeit zwischen 3 und 5 auf `r6`
- RAW Abhängigkeit zwischen 4 und 6 auf `r7`
- RAW Abhängigkeit zwischen 9 und 10 auf `r5`

Wir fügen 5 `NOP` Instruktionen ein, um die Hazards zu beheben:

```
1 LW   r1,    5(r0)
NOP
NOP
2 ADDU r3,    r1,    r2
3 SUBU r6,    r4,    r5
4 LW   r7,    10(r0)
NOP
5 ADDU r8,    r3,    r6
6 BEQ  r7,    r0,    <Ziel>
7 ADDU r9,    r2,    r4
...
Ziel:
8 ADDU r9,    r2,    r5
9 SUBU r5,    r4,    r3
NOP
NOP
10 LW   r10,   5(r5)
```

Zusammen mit den $10 + x$ ursprünglichen Instruktionen sind dies $15 + x$ Instruktionen, die in $(15 + x) \cdot 4 = 60 + (x \cdot 4)$ Byte untergebracht werden können (bei x Instruktionen zwischen 7 und 8).

Unter der Annahme, dass die Verzweigung genommen wird, werden 14 Instruktionen ausgeführt und in 3 Takten wegen eines Verzweigungsbefehls gewartet, dafür werden $14 + 3 + 4 = 21$ Zyklen benötigt.

Mit frühzeitiger (Sprung-)Entscheidung: Bei frühzeitiger (Sprung-)Entscheidung wird nur noch ein Taktzyklus nach einem Branch möglicherweise vergeudet (statt drei). Wir haben also die gleichen Ursachen für Hazards wie zuvor, verschwenden jetzt bei der Ausführung allerdings weniger Zyklen: Unter der Annahme, dass die Verzweigung genommen wird, werden 14 Instruktionen ausgeführt und in einem Takt wegen eines Verzweigungsbefehls gewartet, dafür werden $14 + 1 + 4 = 19$ Zyklen benötigt.

Mit Delay Slot: Mit dem Branch Delay Slot würde Instruktion 7 unabhängig vom Branch ausgeführt werden, hier führt also die Kontrollabhängigkeit zwischen 7 und 6 zu einem Hazard.

Um dies zu beheben, können wir beispielsweise eine `NOP` Instruktion nach Instruktion 6 einfügen:

```
1 LW   r1,    5(r0)
NOP
NOP
2 ADDU r3,    r1,    r2
3 SUBU r6,    r4,    r5
4 LW   r7,    10(r0)
```

```
NOP
5 ADDU r8,    r3,    r6
6 BEQ  r7,    r0,    <Ziel>
NOP
7 ADDU r9,    r2,    r4
...
Ziel:
8 ADDU r9,    r2,    r5
9 SUBU r5,    r4,    r3
NOP
NOP
10 LW   r10,   5(r5)
```

Zusammen mit den $10 + x$ ursprünglichen Instruktionen sind dies $16 + x$ Instruktionen, die in $(16 + x) \cdot 4 = 64 + (x \cdot 4)$ Byte untergebracht werden können (bei x Instruktionen zwischen 7 und 8).

(Alternativ könnte man auch feststellen, dass Instruktion 7, falls sie im Branch Delay Slot belassen wird, im Falle einer genommenen Verzweigung direkt von Instruktion 8 überschrieben werden würde und damit keinen Schaden anrichten könnte. Somit kann diese `NOP` Instruktion im Programmtext eingespart werden.)

Unter der Annahme, dass die Verzweigung genommen wird, werden 15 Instruktionen ausgeführt, dafür werden $15 + 4 = 19$ Zyklen benötigt.

Mit Forwarding: Hier können keine Hazards aufgrund von Datenabhängigkeiten zwischen ALU Operationen auftreten und nur eine Instruktion zwischen einer `load` Instruktion und einer davon lesenden Instruktion ist notwendig.

Aufgrund der Early Branch Resolution muss bei einer RAW Abhängigkeit einer Branch Instruktion zusätzlich eine weitere Instruktion zwischen der schreibenden Instruktion und dem Branch sein.

Somit haben wir nur noch Datenhazards wegen der RAW Abhängigkeiten zwischen 1 und 2 sowie zwischen 4 und 6.

Wir brauchen damit nur noch 3 `NOP` Instruktionen:

```
1 LW   r1,    5(r0)
NOP
2 ADDU r3,    r1,    r2
3 SUBU r6,    r4,    r5
4 LW   r7,    10(r0)
NOP
5 ADDU r8,    r3,    r6
6 BEQ  r7,    r0,    <Ziel>
NOP
7 ADDU r9,    r2,    r4
...
Ziel:
8 ADDU r9,    r2,    r5
9 SUBU r5,    r4,    r3
10 LW   r10,   5(r5)
```

Zusammen mit den $10 + x$ ursprünglichen Instruktionen sind dies $13 + x$ Instruktionen, die in $(13 + x) \cdot 4 = 52 + (x \cdot 4)$ Byte untergebracht werden können (bei x Instruktionen zwischen 7 und 8).

Unter der Annahme, dass die Verzweigung genommen wird, werden 12 Instruktionen ausgeführt, dafür werden $12 + 4 = 16$ Zyklen benötigt.

Mit Stalling: Durch (Daten-)Stalling kann die Hardware mit sämtlichen Datenhazards umgehen, daher brauchen wir hier nur noch die `NOP` Instruktion für den Branch Delay Slot:

```
1 LW   r1,    5(r0)
```

```
2 ADDU r3,    r1,    r2
3 SUBU r6,    r4,    r5
4 LW   r7,    10(r0)
5 ADDU r8,    r3,    r6
6 BEQ  r7,    r0,    <Ziel>
NOP
7 ADDU r9,    r2,    r4
...
Ziel:
8 ADDU r9,    r2,    r5
9 SUBU r5,    r4,    r3
10 LW   r10,   5(r5)
```

Zusammen mit den $10 + x$ ursprünglichen Instruktionen sind dies $11 + x$ Instruktionen, die in $(11 + x) \cdot 4 = 44 + (x \cdot 4)$ Byte untergebracht werden können (bei x Instruktionen zwischen 7 und 8).

Die Ausführungsdauer verändert sich jedoch nicht gegenüber der Variante ohne Stalling.

Der Vollständigkeit halber: Mit aggressiverem Umsortieren der Instruktionen lassen sich schon in der Variante mit Forwarding die Instruktionen ohne zusätzliche `NOP` Instruktionen bei gleich bleibender Semantik ausführen:

```
1 LW   r1,    5(r0)
4 LW   r7,    10(r0)
2 ADDU r3,    r1,    r2
3 SUBU r6,    r4,    r5
6 BEQ  r7,    r0,    <Ziel>
5 ADDU r8,    r3,    r6
7 ADDU r9,    r2,    r4
...
Ziel:
8 ADDU r9,    r2,    r5
9 SUBU r5,    r4,    r3
10 LW   r10,   5(r5)
```

Hier werden nur $10 + x$ Instruktionen mit jeweils 4 Byte benötigt.
Bei genommener Verzweigung werden nur 9 Instruktionen ausgeführt, diese benötigen $9 + 4 = 13$ Takte.

Aufgabe 9.3: Hazard Unit

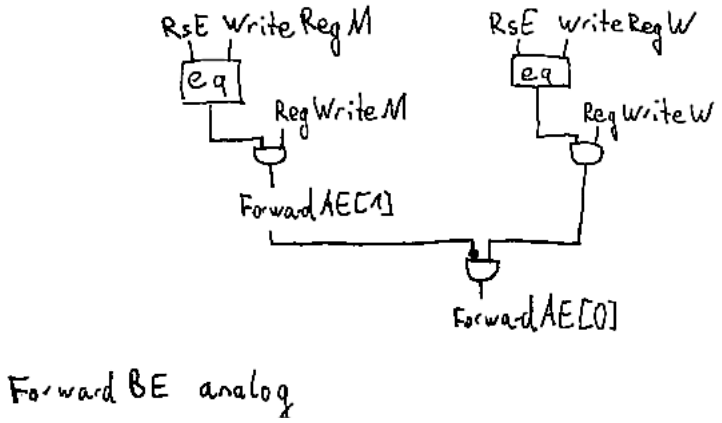

Abbildung 2: Teil 1

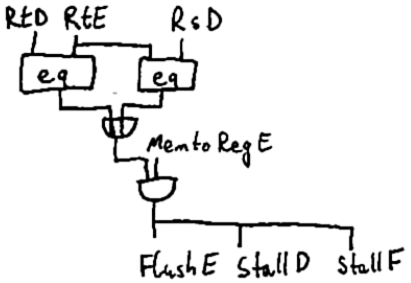

Abbildung 3: Teil 2

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Lucas Biehl

System Architecture SS 2021

Solution Sketch 9

Problem 9.1: Pipelining (1)

Violated assumptions:

Homogeneous partial operations: Certain sub-operations cannot be further divided, or only with a very large effort, for example memory accesses or complex arithmetic operations. For example, clocked registers would have to be inserted *inside* the memory/arithmetic circuit.

For certain types of memory, there are also physical processes that cannot be further divided or parallelized (for example, the movement of a hard disk head or the reading of a bit, i.e., reading the charge of a capacitor in DRAM).

Thus, the latency of some sub-operations is bounded from below, and if the other operations are further divided for a multi-stage pipeline, the latencies of the various steps would become non-homogeneous.

Repetition of independent operations: Violated anyway and only solvable by a hazard unit or software constraints (losing parts of the idealized speedup). With 100 stages there are *yet* more possibilities for logically interdependent operations.

Problem 9.2: Pipelining (2)

The data and control dependencies are shown in Figure 4.

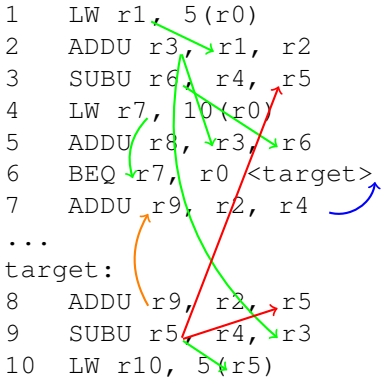

Figure 4: Data and control dependencies: RAW, WAW, WAR, and Control

In the following, we assume that the general-purpose register file writes on falling clock edges.

Without additional hardware: Pipelining can only violate RAW and control dependencies, as the order of write accesses to each other is not changed.

The following points should be noted here:

- After a branch instruction, the next 3 clocks may not be used for relevant computations (due to stalling or branch prediction).
- There must be at least 2 instructions between an instruction that writes to register `rx` and a subsequent instruction that reads from `rx`.

With this, we can identify the following causes of hazards:

- RAW dependency between 1 and 2 on `r1`
- RAW dependency between 3 and 5 on `r6`
- RAW dependency between 4 and 6 on `r7`
- RAW dependency between 9 and 10 on `r5`

We insert 5 `NOP` instructions to fix the hazards:

```
1 LW   r1,    5(r0)
NOP
NOP
2 ADDU r3,    r1,    r2
3 SUBU r6,    r4,    r5
4 LW   r7,    10(r0)
NOP
5 ADDU r8,    r3,    r6
6 BEQ  r7,    r0,    <target>
7 ADDU r9,    r2,    r4
...
target:
8 ADDU r9,    r2,    r5
9 SUBU r5,    r4,    r3
NOP
NOP
10 LW   r10,   5(r5)
```

Together with the $10+x$ original instructions, we now have $15+x$ instructions that can be stored in $(15+x)\cdot 4 = 60 + (x \cdot 4)$ bytes (where x is the number of instructions between 7 and 8).

Assuming that the branch is taken, 14 instructions are executed and in 3 cycles there is a delay because of a branch instruction; this requires $14 + 3 + 4 = 21$ cycles.

With early branch resolution: With early branch resolution, only one clock cycle is potentially wasted after a branch (instead of three). So we have the same causes of hazards as before, but now waste fewer cycles during execution: Assuming that the branch is taken, 14 instructions are executed and in one cycle there is a delay because of a branch instruction; this requires $14 + 1 + 4 = 19$ cycles.

With delay slot: With the branch delay slot, instruction 7 would be executed independently of the branch; so here the control dependency between 7 and 6 leads to a hazard.

To fix this, we can, for example, insert a `NOP` instruction after instruction 6:

```
1 LW   r1,    5(r0)
NOP
NOP
2 ADDU r3,    r1,    r2
3 SUBU r6,    r4,    r5
4 LW   r7,    10(r0)
NOP
```

```
5 ADDU r8,    r3,    r6
6 BEQ  r7,    r0,    <target>
NOP
7 ADDU r9,    r2,    r4
...
target:
8 ADDU r9,    r2,    r5
9 SUBU r5,    r4,    r3
NOP
NOP
10 LW   r10,   5(r5)
```

Together with the $10+x$ original instructions, we now have $16+x$ instructions that can be stored in $(16+x)\cdot 4 = 64 + (x \cdot 4)$ bytes (where x is the number of instructions between 7 and 8).

(Alternatively, one might note that instruction 7, if left in the branch delay slot, would be overwritten directly by instruction 8 in the case of a taken branch, and thus could do no harm. Thus the corresponding `NOP` instruction can be omitted).

Assuming that the branch is taken, 15 instructions are executed; this requires $15 + 4 = 19$ cycles.

With forwarding: Here, no hazards due to data dependencies between ALU operations can occur, and only one instruction between a `load` instruction and a dependent read instruction is necessary.

Due to the early branch resolution, if there is a RAW dependency of a branch instruction, there must be an additional instruction between the writing instruction and the branch.

Thus, we have only data hazards because of the RAW dependencies between 1 and 2, and between 4 and 6.

Therefore, we need only 3 `NOP` instructions:

```
1 LW   r1,    5(r0)
NOP
2 ADDU r3,    r1,    r2
3 SUBU r6,    r4,    r5
4 LW   r7,    10(r0)
NOP
5 ADDU r8,    r3,    r6
6 BEQ  r7,    r0,    <target>
NOP
7 ADDU r9,    r2,    r4
...
target:
8 ADDU r9,    r2,    r5
9 SUBU r5,    r4,    r3
10 LW   r10,   5(r5)
```

Together with the $10+x$ original instructions, we now have $13+x$ instructions that can be stored in $(13+x)\cdot 4 = 52 + (x \cdot 4)$ bytes (where x is the number of instructions between 7 and 8).

Assuming that the branch is taken, 12 instructions are executed; this requires $12 + 4 = 16$ cycles.

With stalling: (Data) stalling allows the hardware to handle all data hazards, so all we need here is the `NOP` instruction for the branch delay slot:

```
1 LW   r1,    5(r0)
2 ADDU r3,    r1,    r2
3 SUBU r6,    r4,    r5
4 LW   r7,    10(r0)
5 ADDU r8,    r3,    r6
6 BEQ  r7,    r0,    <target>
```

```
NOP
7 ADDU r9,    r2,    r4
...
target:
8 ADDU r9,    r2,    r5
9 SUBU r5,    r4,    r3
10 LW   r10,   5(r5)
```

Together with the $10+x$ original instructions, we now have $11+x$ instructions that can be stored in $(11+x)\cdot 4 = 44 + (x \cdot 4)$ bytes (where x is the number of instructions between 7 and 8).

However, the execution time does not change compared to the variant without stalling.

For completeness: With more aggressive reordering of instructions, even in the variant with forwarding, the instructions can be executed without additional `NOP` Instructions while keeping the semantics the same:

```
1 LW   r1,    5(r0)
4 LW   r7,    10(r0)
2 ADDU r3,    r1,    r2
3 SUBU r6,    r4,    r5
6 BEQ  r7,    r0,    <target>
5 ADDU r8,    r3,    r6
7 ADDU r9,    r2,    r4
...
target:
8 ADDU r9,    r2,    r5
9 SUBU r5,    r4,    r3
10 LW   r10,   5(r5)
```

Here, we only need $10+x$ instructions with 4 bytes each. Assuming that the branch is taken, only 9 instructions are executed; these require $9 + 4 = 13$ cycles.

Problem 9.3: Hazard Unit

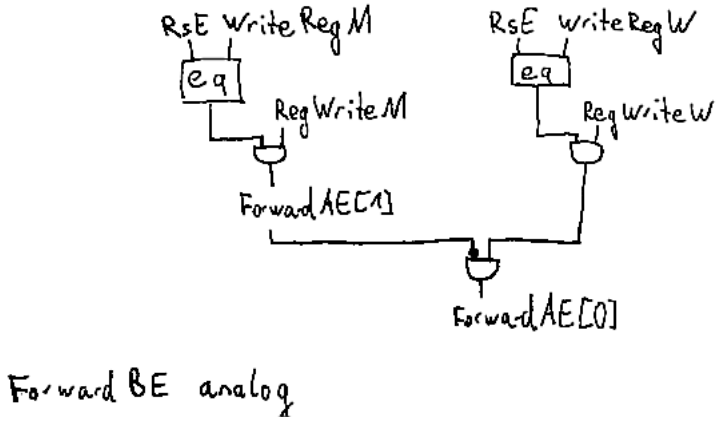

Figure 5: Part 1

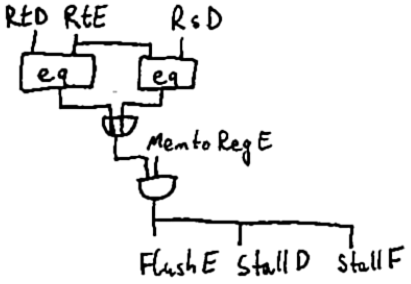

Figure 6: Part 2