Übung 8

Abgabe bis **Donnerstag, 9. Juni 10:00** via EPIIC: `http://ep.iic.jku.at`.

SystemVerilog Dateien müssen vor der Abgabe in EPIIC ausgeführt werden. Dateien, die nicht kompilieren können nicht abgegeben werden.

1. **Pipelining Implementierung (10 Punkte)**

   Erweitere den MIPS Prozessor aus Übung 7[1] um eine fünfstufige Pipeline (siehe Abbildung 1). Erweitere auch die Kontrolleinheit, sodass korrekte Signale generiert werden wenn Register der Pipeline zurückgesetzt werden. Teste deine Implementierung, wie in Übung 7, mit dem Modul *testbench* aus der Datei *test_sum.sv*. Die Dateien *test_sum.sv* und *sum.s* wurden angepasst um die Pipeline-Konflikte zu lösen.

   Hinweis: Im Modul *register* muss `or posedge reset` gelöscht werden, damit die Pipeline richtig zurückgesetzt werden kann.

2. **Datenabhängigkeiten (6 Punkte)**

   Unter der Annahme, dass in einem Programm Befehl S_i vor Befehl S_j ausgeführt wird, kann man Datenabhängigkeiten folgendermaßen klassifizieren:

   - *True Dependence* (read after write): S_j liest eine Variable, die von S_i geschrieben wird.
   - *Anti Dependence* (write after read): S_j schreibt eine Variable, die in S_i gelesen wird.
   - *Output Dependence* (write after write): S_j schreibt eine Variable, die auch von S_i geschrieben wird.

   Finde und klassifiziere alle Datenabhängigkeiten des folgenden MIPS-Programms:

```
s1      addi $s1, $s0, 2
   loop:
s2      beq  $s1, $0, endloop
s3          add  $s0, $s2, $s2
s4          add  $s2, $s3, $s4
s5          and  $s3, $s1, $s3
s6          xor  $s5, $s2, $s1
s7          addi $s4, $s2,  4
s8          addi $s1, $s1, -1
s9          j    loop
   endloop:
s10     or   $s4, $s4, $s2
s11     addi $s6, $s6, 9
s12     sub  $s5, $s6, $s1
```

[1] Nach der Abgabe von Übung 7 wird die Musterlösung zur Verfügung gestellt, damit die Aufgabe unabhängig von Übung 7 lösbar ist.

3. Pipeline-Konflikte (8 Punkte)

Betrachte noch einmal das Programmstück aus Aufgabe 2. Die Befehle werden in einer fünfstufigen Pipeline mit den Stufen *fetch*, *decode*, *execute*, *memory* und *writeback* abgearbeitet (siehe Abbildung 1). Schreibt ein Befehl in ein Register, so ist der geschriebene Wert erst nach der *writeback* Stufe verfügbar.

(a) Finde und begründe alle auftretenden Pipeline-Konflikte.

(b) Behebe alle Pipeline-Konflikte durch Einfügen einer minimalen Anzahl an NOP-Befehlen.

(c) Minimiere die Anzahl der benötigten NOP-Befehle durch geschicktes Umordnen der Befehle. Die Semantik des Programms darf dabei nicht verändert werden.

(d) Was würde sich an den Lösungen aus 3.b und 3.c ändern wenn der Prozessor ein Rücksetzen der Register nicht unterstützten würde?

Hinweis: Das Programm liegt auch hexadezimal kodiert vor, sodass du deine Lösung auf dem Prozessor aus Aufgabe 1 ausführen und damit testen kannst[2].

4. (Bonus) Laufzeiten (4 Punkte)

Betrachte das Programmstück aus Aufgabe 3.b und nimm an, dass das Register `$s0` den Wert 10 beinhaltet und die Prozessorkomponenten die Verzögerungen, wie in Tabelle 1 angegeben haben (Für nicht gelistete Elemente kann eine Verzögerung von $0\,ps$ angenommen werden).

(a) Berechne die Ausführungszeit des Programms auf dem MIPS-Prozessor ohne Pipelining (siehe Übung 7).

(b) Berechne die Ausführungszeit des Programms auf dem MIPS-Prozessor mit einer fünfstufigen Pipeline (siehe Abbildung 1).

(c) Wie verändert sich die Laufzeit aus Aufgabe 4.b unter Annahme dass ein Branch-Predictor in der Stufe *decode* annimmt, dass Sprünge immer getätigt werden? Wie muss dazu der Datenpfad aus Abbildung 1 geändert werden?

Tabelle 1: Verzögerungen der Prozessorkomponenten

Komponente	Verzögerung [*ps*]
ALU	200
Instruction Memory	250
Data Memory	250
Register File	150

[2]Beim Einfügen eines Befehls zwischen Sprung und Sprungmarke muss der hexadezimalwert des `beq` Befehls um 1 erhöht werden. Analog gilt das für den Befehl `j` wenn Befehle vor der Sprungmarke eingefügt werden.

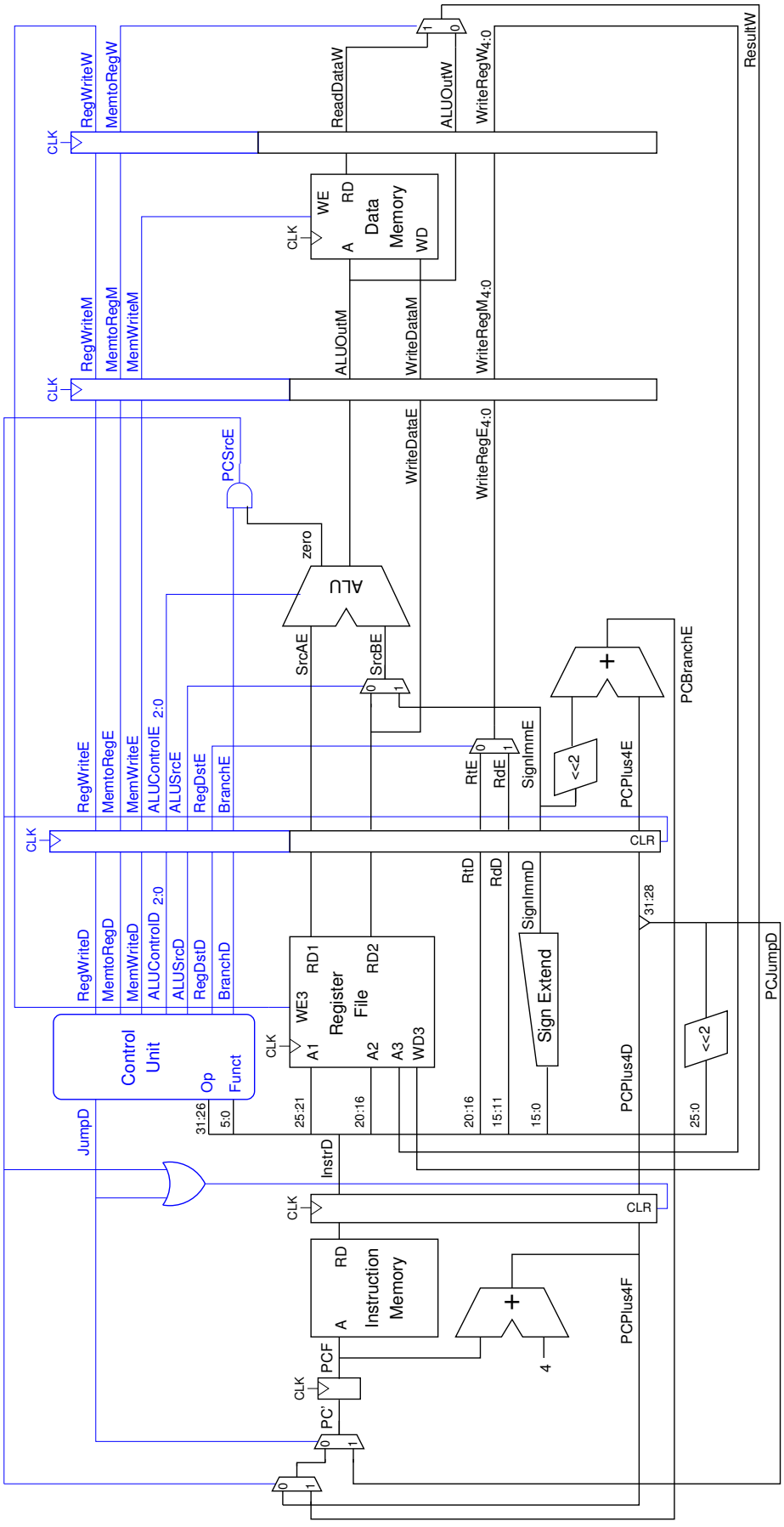

Abbildung 1: MIPS Prozessor mit fünfstufiger Pipeline