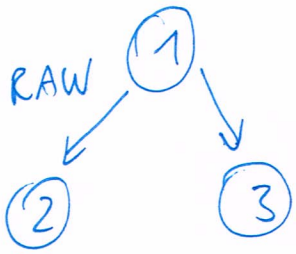
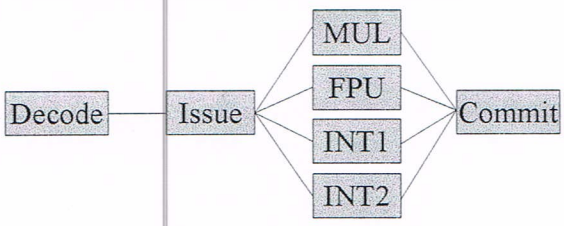
1. Abhängigkeitsgraph, Reorder Buffer

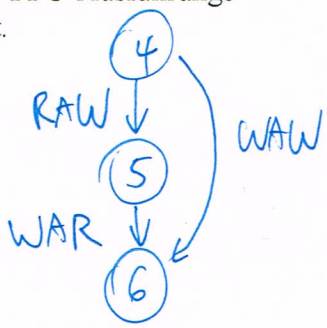


Gegeben sind die folgenden sechs Befehle:

```
1    fmul R1, R2, R3          ; R1 := R2 * R3
2    fadd R4, R1, R2          ; R4 := R1 + R2
3    mov  R5, R1              ; R5 := R1
4    fmul R6, R2, R2          ; R6 := R2 * R2
5    fadd R8, R6, R6          ; R8 := R6 * R6
6    mov  R6, R7              ; R6 := R7
```

Die Befehle `fmul` und `fadd` benötigen jeweils zwei Zyklen auf der MUL- bzw. FPU-Ausführungseinheit, der Befehl `mov` benötigt einen Zyklus auf einer INT-Ausführungseinheit.





a) Zeichnen Sie den Abhängigkeitsgraphen der sechs Befehle.

b) Die Befehle werden auf einem System mit Reorder Buffer (wie im Beispiel aus dem Lehrbrief; $n = 4$, $in_{max} = out_{max} = 2$) ausgeführt. Erstellen Sie die tabellarischen Übersichten des Reorder Buffers für die Zeitpunkte t_0 bis t_6. (Zum Zeitpunkt t_7 sollte der Puffer leer sein!)

$t = 0$			Operanden				
Befehl	**Einh.**	**Status**	**R:a**	**R:b**	**I:a**	**I:b**	**Ergebnis**
1 – fmul	MUL	▶ 1	R2	R3			R1 (u.)
2 – fadd	FPU	"	—	R2	1-fmul		R4 (u.)

$t = 1$			Operanden				
Befehl	**Einh.**	**Status**	**R:a**	**R:b**	**I:a**	**I:b**	**Ergebnis**
1 – fmul	MUL	▶ 2					R1 (u.)
2 – fadd	FPU	"	—	R2	1-fmul		R4 (u)
3 – mov	INT1	"	—		1-fmul		R5 (u)

$t = 2$			Operanden				
Befehl	**Einh.**	**Status**	**R:a**	**R:b**	**I:a**	**I:b**	**Ergebnis**
1 – fmul	—	✓					R1
2 – fadd	FPU	▶ 1	R1*	R2			R4 (u)
3 – mov	INT1	▶	R1*				R5 (u)
4 – fmul	MUL	▶ 1	R2	R2			R6 (u)

$t = 3$			Operanden				
Befehl	Einh.	Status	R:a	R:b	I:a	I:b	Ergebnis
2 – fadd	FPU	▶2					R4 (u)
3 – mov	—	✓					R5
4 – fmul	MUL	▶2					R6 (u)

$t = 4$			Operanden				
Befehl	Einh.	Status	R:a	R:b	I:a	I:b	Ergebnis
2 – fadd	—	✓					R4
3 – mov	—	✓					R5
4 – fmul	—	✓					R6
5 – fadd	FPU	▶1	R6*	R6*			R8 (u)

$t = 5$			Operanden				
Befehl	Einh.	Status	R:a	R:b	I:a	I:b	Ergebnis
4 – fmul	—	✓					R6
5 – fadd	FPU	▶2					R8 (u)
6 – mov	INT1	▶	R7				R6 (u)

$t = 6$			Operanden				
Befehl	Einh.	Status	R:a	R:b	I:a	I:b	Ergebnis
5 – fadd	—	✓					R8
6 – mov	—	✓					R6

$t = 7$: Puffer leer

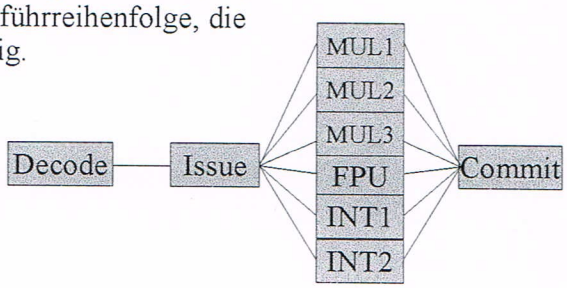
2. Instruction Scheduling

Compiler können beim Erzeugen eines Binarys (also einer ausführbaren Maschinensprachedatei) die Reihenfolge von Assembler-Instruktionen vertauschen, um für ein effizienteres Befüllen der Pipeline zu sorgen – z. B. mit dem Ziel, Wartezeiten wegen belegter Ausführeinheiten zu vermeiden.

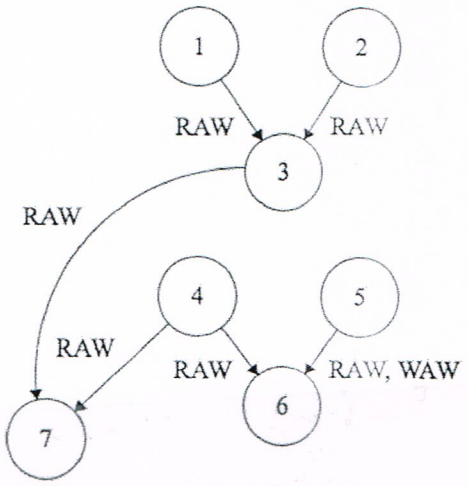
Bei dieser Umsortierung, die *instruction scheduling* genannt wird (siehe dazu auch die Wikipedia-Seite https://en.wikipedia.org/wiki/Instruction_scheduling), müssen die bekannten Abhängigkeiten (RAW, WAR, WAW) beachtet werden, damit der umsortierte Code bei der Ausführung zum selben Ergebnis wie die Ausführung des Codes in Originalreihenfolge führt.

Technisch wird das gelöst, indem ein Abhängigkeitsgraph erzeugt und dazu eine *topologische Sortierung* (vgl. GDI 3) erstellt wird. Jede Ausführreihenfolge, die einer solchen topologischen Sortierung entspricht, ist zulässig.

Betrachten Sie dazu das folgende Beispiel, das auf der Maschine aus Aufgabe 1 (**aber mit drei MUL-Einheiten MUL1, MUL2, MUL3 und beschleunigtem Commit via $out_{max} = 4$**) durch Umsortieren beschleunigt ausgeführt werden soll:



```
1    fadd R5, R1, R2
2    fsub R6, R1, R2
3    fmul R7, R5, R6
4    mov  R8, R1
5    mov  R9, R2
6    fmul R9, R8, R9
7    fmul R3, R8, R7
```



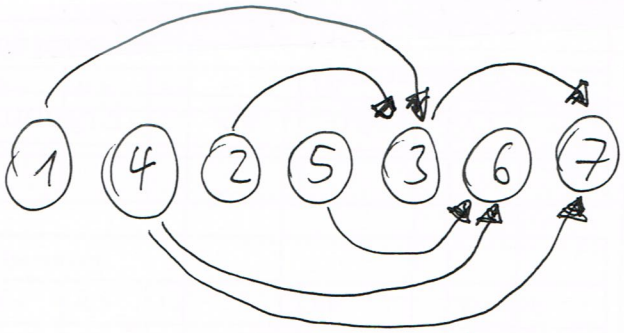
Der Abhängigkeitsgraph zu diesem Code-Ausschnitt ist auf der rechten Seite abgedruckt.

Bei der Ausführung ergibt sich das Problem, dass die Befehle 1: `fadd` und 2: `fsub` beide die FPU-Einheit benötigen, die nur 1x vorhanden ist. Es entsteht damit:

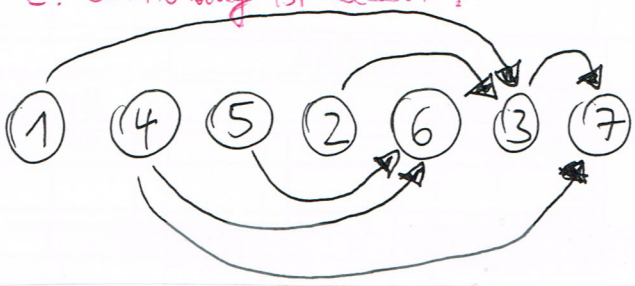
$t = 0$		
Befehl	**Einh.**	**Status**
`fadd`	FPU	▶1

$t = 1$		
Befehl	**Einh.**	**Status**
`fadd`	FPU	▶2

$t = 2$		
Befehl	**Einh.**	**Status**
`fadd`	–	✓
`fsub`	FPU	▶1



2. Sortierung ist besser!



(Der Buffer hat immer vier Zeilen; oben wurden die untersten zwei Zeilen weggelassen, um Platz zu sparen.)

Die Umsortierung 1, 6, 2, 7, 3, 4, 5 (siehe rechter Rand) würde den Ablauf beschleunigen, ist aber nicht zulässig, weil Befehl 7 von Befehl 3 abhängt (3 $<_{\text{RAW}}$ 7). Finden Sie eine topologische Sortierung und geben Sie das daraus resultierende Programm an. Es soll auf der Maschine schneller ausgeführt werden, was Sie daran erkennen können, dass im Reorder Buffer immer zwei Befehle aktiv sind.

Füllen Sie dazu für die Zeitpunkte t=0 bis t=6 die Tabellen aus:

```
1    fadd R5, R1, R2
6    fmul R9, R8, R9
2    fsub R6, R1, R2
7    fmul R3, R8, R7
3    fmul R7, R5, R6
4    mov  R8, R1
5    mov  R9, R2
```

$t = 0$			Operanden				
Befehl	**Einh.**	**Status**	**R:a**	**R:b**	**I:a**	**I:b**	**Ergebnis**
1 – fadd	FPU	▶1	R1	R2			R5 (u)
4 – mov	INT	▶1	R1				R8 (u)

$t = 1$			Operanden				
Befehl	Einh.	Status	R:a	R:b	I:a	I:b	Ergebnis
1 - fadd	FPU	▶2					R5 (u)
4 - mov	—	✓					R8
5 - mov	INT1	▶	R2				R9 (u)

$t = 2$			Operanden				
Befehl	Einh.	Status	R:a	R:b	I:a	I:b	Ergebnis
1 - fadd	—	✓					R5
4 - mov	—	✓					R8
5 - mov	—	✓					R9
2 - fsub	FPU	▶1	R1	R2			R6 (u)

$t = 3$			Operanden				
Befehl	Einh.	Status	R:a	R:b	I:a	I:b	Ergebnis
2 - fsub	FPU	▶2					R6 (u)
6 - fmul	MUL1	▶1	R8	R9			R9 (u)
3 - fmul	MUL2	‖	R5			2-fsub	R7 (u)

$t = 4$			Operanden				
Befehl	Einh.	Status	R:a	R:b	I:a	I:b	Ergebnis
2 - fsub	—	✓					R6
6 - fmul	MUL1	▶2					R9 (u)
3 - fmul	MUL2	▶1	R5	R6*			R7 (u)
7 - fmul	MUL3	‖	R8			3-fmul	R3 (u)

$t = 5$			Operanden				
Befehl	Einh.	Status	R:a	R:b	I:a	I:b	Ergebnis
6 - fmul	—	✓					R9
3 - fmul	MUL2	▶2					R7 (u)
7 - fmul	MUL3	‖	R8			3-fmul	R3 (u)

$t = 6$			Operanden				
Befehl	Einh.	Status	R:a	R:b	I:a	I:b	Ergebnis
3 - fmul	—	✓					R7
7 - fmul	MUL3	▶1	R8	R7*			

t=7: 7-fmul / ▶2 t=8: 7-fmul / ✓ t=9: Buffer leer