Rechnerarchitektur (RA)

Sommersemester 2020

Instruction Level Parallelism (ILP)

Jian-Jia Chen
Informatik 12
http://ls12-www.cs.tu-dortmund.de/daes/

Instruction Level Parallelism

Instruction-Level Parallelism (ILP): Die Ausführung der Befehle überlappt sich um Perfomanz zu verbessern.

- Zwei Möglichkeiten

1) softwarebasiert: es basiert auf Software Technologie, die Parallelität wird während der Kompilierung gefunden und ausgenutzt. Es wird *statische ILP* genannt

2) hardwarebasiert: es basiert auf Hardware Technologie, die Parallelität wird während der Ausführung gefunden und ausgenutzt. Es wird *dynamische ILP* genannt

Software Technologie: Beispiel

Beispiel: Die Elemente des Vektors und eine Konstante (hier *s*) addieren

```
for (i=1000; i>0; i=i-1)
      x[i] = x[i] + s;
```

Annahme: die Latenz für alle folgenden Beispiele

Befehl erzeugt Ergebnis	*Befehl benutzt Ergebnis*	*Latenz (Zyklen)*	*stalls zwischen (Zyklen)*
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	2	1
Load double	Store double	1	0
Integer op	Integer op	1	0

FP Schleife: Wo sind die Hazards (Gefährdungen)?

- Zuerst übersetzen wir das Programm in MIPS:
  - Für eine vereinfachte Berechnung nehmen wir an, dass jedes Element des Vektors 8 Bytes groß ist.

```
Loop:   L.D     F0,0(R1)  ;F0=vector element
        ADD.D   F4,F0,F2  ;add scalar from F2
        S.D     0(R1),F4  ;store result
        DADDUI  R1,R1,-8  ;decrement pointer 8B(DW)
        BNEZ    R1,Loop   ;branch R1!=zero
```

FP Schleife: die Stalls

```
1 Loop: L.D    F0,0(R1)  ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2  ;add scalar in F2
4       stall
5       stall
6       S.D    0(R1),F4  ;store result
7       DADDUI R1,R1,-8  ;decrement pointer 8B (DW)
8       stall            ;assumes can't forward to branch
9       BNEZ   R1,Loop   ;branch R1!=zero
```

Befehl erzeugt Ergebnis	*Befehl benutzt Ergebnis*	*Latenz (Zyklen)*	*stalls zwischen (Zyklen)*
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	2	1
Load double	Store double	1	0
Integer op	Integer op	1	0

9 Taktzyklen: Können wir die Stalls reduzieren, um die Performanz des Programms zu verbesseren?

FP Schleife mit minimierten Stalls

```
1 Loop: L.D    F0,0(R1)
2       DADDUI R1,R1,-8
3       ADD.D  F4,F0,F2
4       stall
5       stall
6       S.D    8(R1),F4 ;altered offset when move DSUBUI
7       BNEZ   R1,Loop
```

DADDUI wird direkt nach L.D ausgeführt, und die Zieladresse des Befehls S.D ist R1+8.

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

7 Taktzyklen, aber nur für 3 effektive Befehle (L.D, ADD.D,S.D), 4 für die Kontrolle der Schleife. Wie können wir die Schleife weiter verbessern?

Aufgelöste Schleife (noch nicht optimiert)
Unrolled Loop

```
1  Loop:L.D    F0,0(R1)
3       ADD.D  F4,F0,F2
6       S.D    0(R1),F4      ;drop DSUBUI & BNEZ
7       L.D    F6,-8(R1)
9       ADD.D  F8,F6,F2
12      S.D    -8(R1),F8     ;drop DSUBUI & BNEZ
13      L.D    F10,-16(R1)
15      ADD.D  F12,F10,F2
18      S.D    -16(R1),F12   ;drop DSUBUI & BNEZ
19      L.D    F14,-24(R1)
21      ADD.D  F16,F14,F2
24      S.D    -24(R1),F16
25      DADDUI R1,R1,#-32    ;alter to 4*8
26      stall
27      BNEZ   R1,LOOP
```

1 cycle stall

2 cycles stall

27 Taktzyklen, oder 6,75 pro Iteration

(Annahme: die Größe der Schleife ist ein ganzzahliges Vielfaches von 4)

Aufgelöste Schleife (optimiert)
Unrolled Loop

```
1 Loop:L.D    F0,0(R1)
2      L.D    F6,-8(R1)
3      L.D    F10,-16(R1)
4      L.D    F14,-24(R1)
5      ADD.D  F4,F0,F2
6      ADD.D  F8,F6,F2
7      ADD.D  F12,F10,F2
8      ADD.D  F16,F14,F2
9      S.D    0(R1),F4
10     S.D    -8(R1),F8
11     S.D    -16(R1),F12
12     DSUBUI R1,R1,#32
13     S.D    8(R1),F16 ; 8-32 = -24
14     BNEZ   R1,LOOP
```

14 Taktzyklen, oder 3,5 pro Iteration

Aufgelöste Schleife: Detail

- Die Grenze der Schleife: *unbekannt* meistens
- Annahme: es ist n, und wir möchten die Schleife auslösen. Jede Iteration der Schleife hat k Kopien von der originellen Schleife (k ist 4 in das frühere Beispiel)
- Wir sollen ein paar konsekutive Schleife generieren:
  - Zuerst führen wir (n mod k) Iterationen mit der originellen Schleife aus
  - Danach führen wir (n/k) Iterationen mit der aufgelösten Schleife aus
- Wenn n gross genug ist, wird die meiste Ausführungszeit in der aufgelösten Schleife verbracht und wird die Performanz verbessert

5 Entscheidungen für Aufgelöste Schleifen

1. die unabhängigen Befehle einer Schleife finden

2. die verschiedenen Register benutzen, um die unnötigen Datenabhängigkeiten zu vermeiden

3. Die zusätzlichen Sprungbefehle eliminieren und die Terminationsbefehle und Iterationsbefehle der Schleife anpassen

4. Die austauschbaren Ladenbefehle und Speicherbefehle determinieren

5. Die Ausführungsfolge der Befehle anordnen, sodass das Ergebnis gleich wie bei der originellen Ausführungsfolge ist

Limitierungen der Aufgelösten Schleifen

1. Je mehr aufgelöste Schleifen, desto weniger amortisierte Kosten der Kontrolle der Schleifen
   - Amdahles Gesetz

2. Wachstum der Programmgröße
   - Eventuell höher „miss rate“ bei Befehlscache

3. Die Grenze der Zahl der Register
   - Die Vorteile der aufgelösten Schleifen sind verloren, wenn nicht alle aktive Werte auf einzigen Register zugeordnet werden können.

Aufgelöste Schleifen können auch den Einfluss auf Sprungbefehle (von pipeline) reduzieren

Dynamisches Scheduling von Befehlen
- in-order execution -

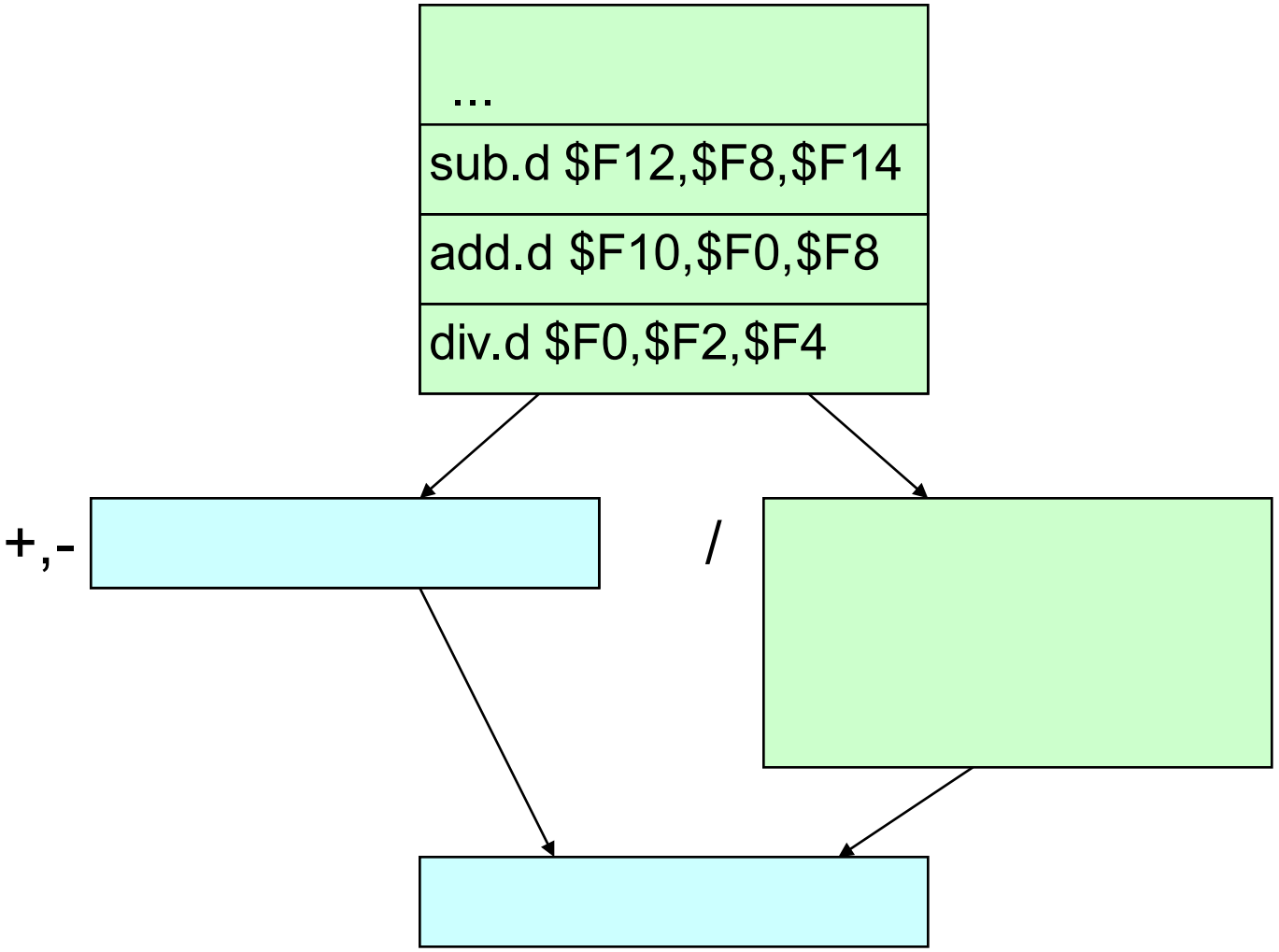
Bislang: Reihenfolge der Befehlsbearbeitung =

Reihenfolge der Befehle im Speicher, abgesehen von Sprüngen; behindert schnelle Ausführung.

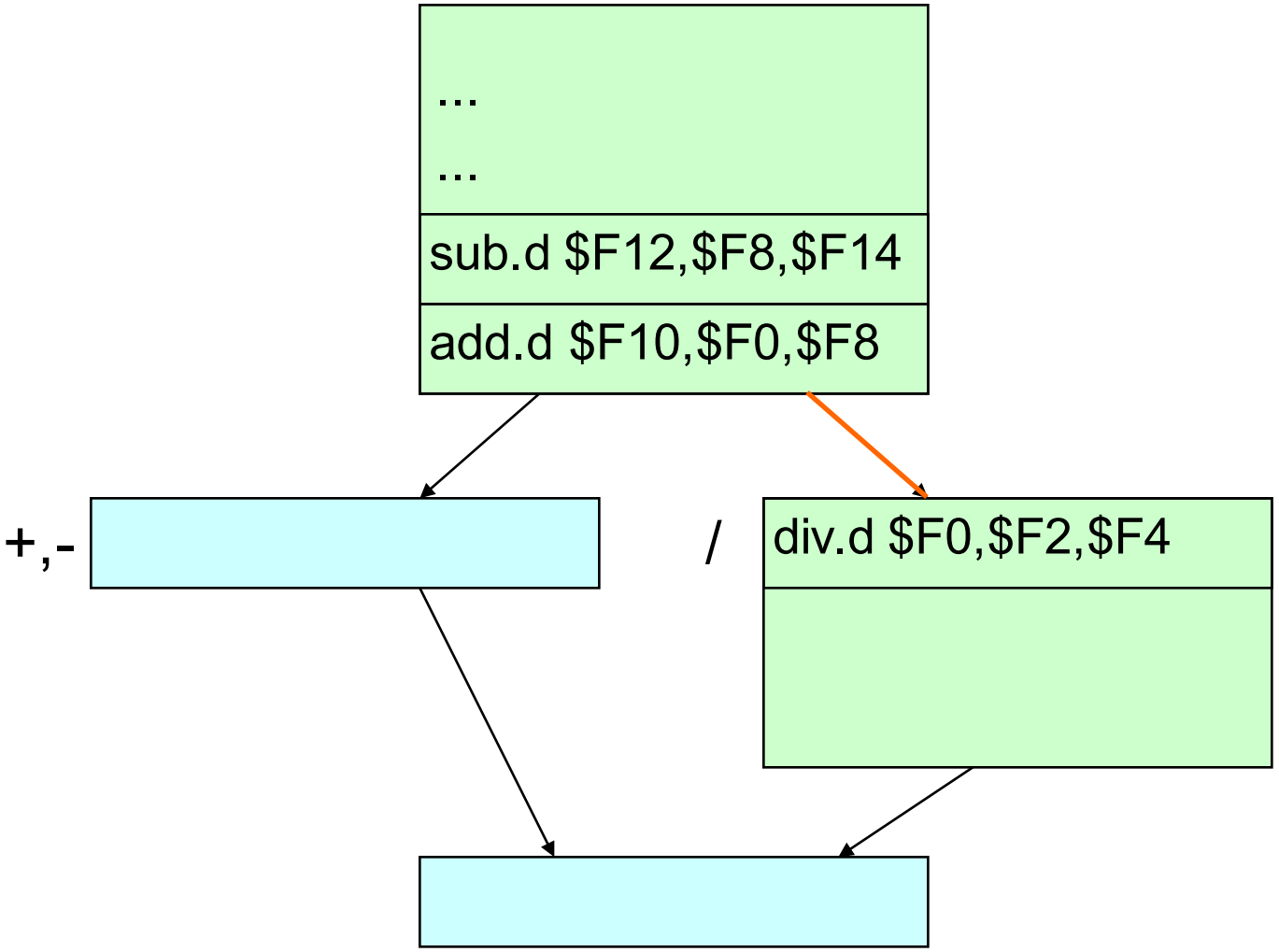
Beispiel:

sub.d $F12,$F8,$F14
add.d $F10,$F0,$F8
div.d $F0,$F2,$F4

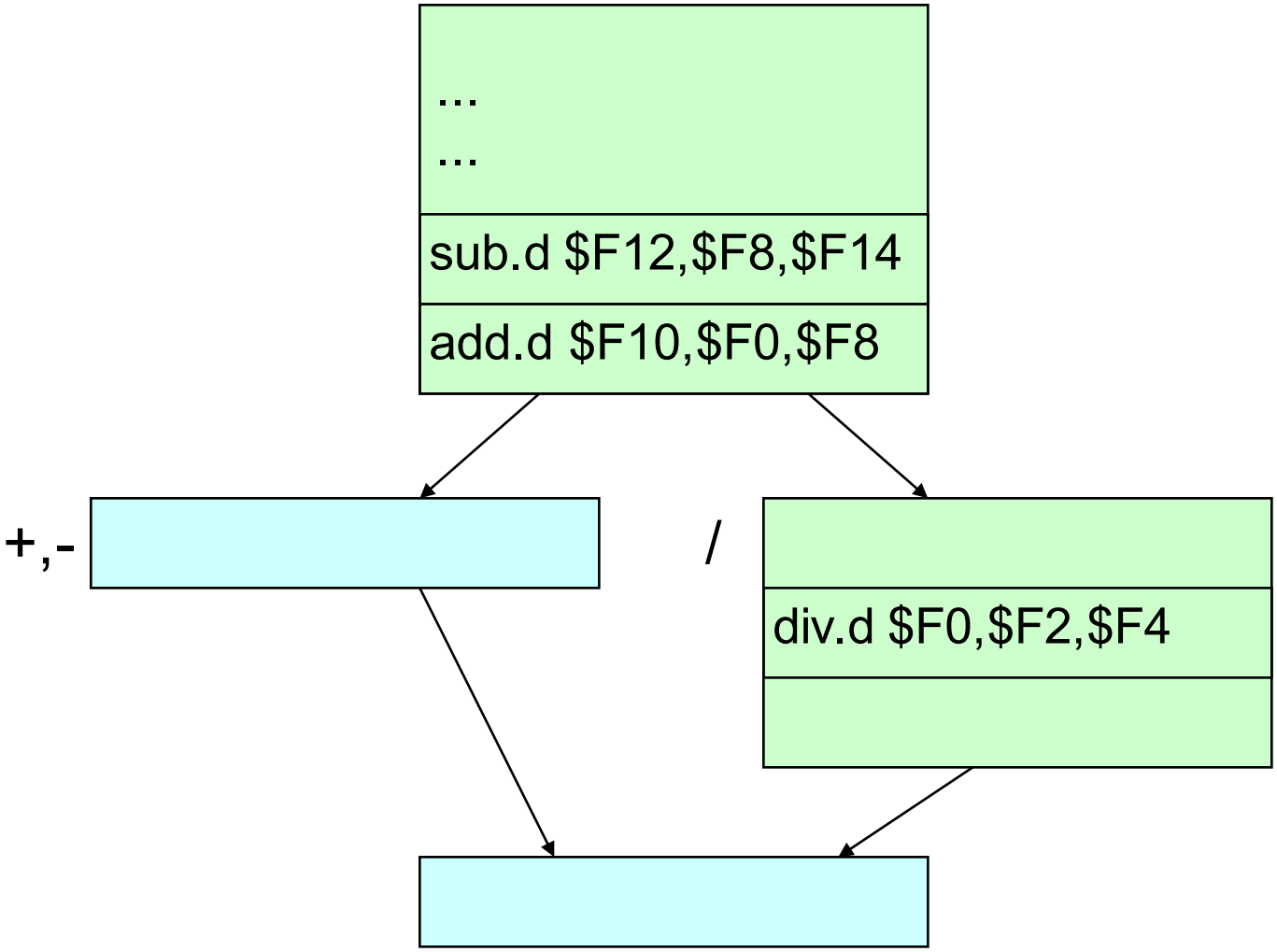
- in-order execution -



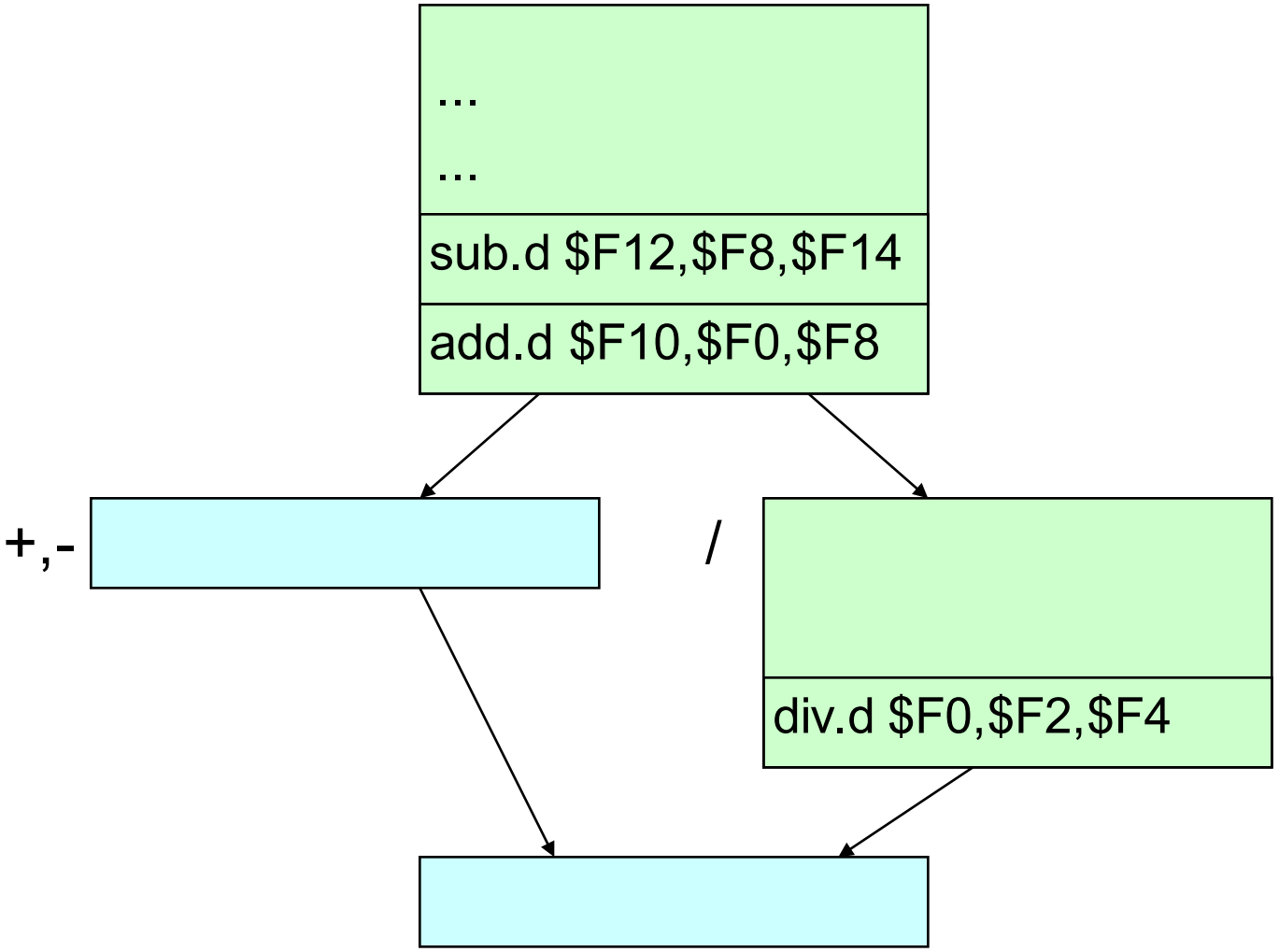
- in-order execution -



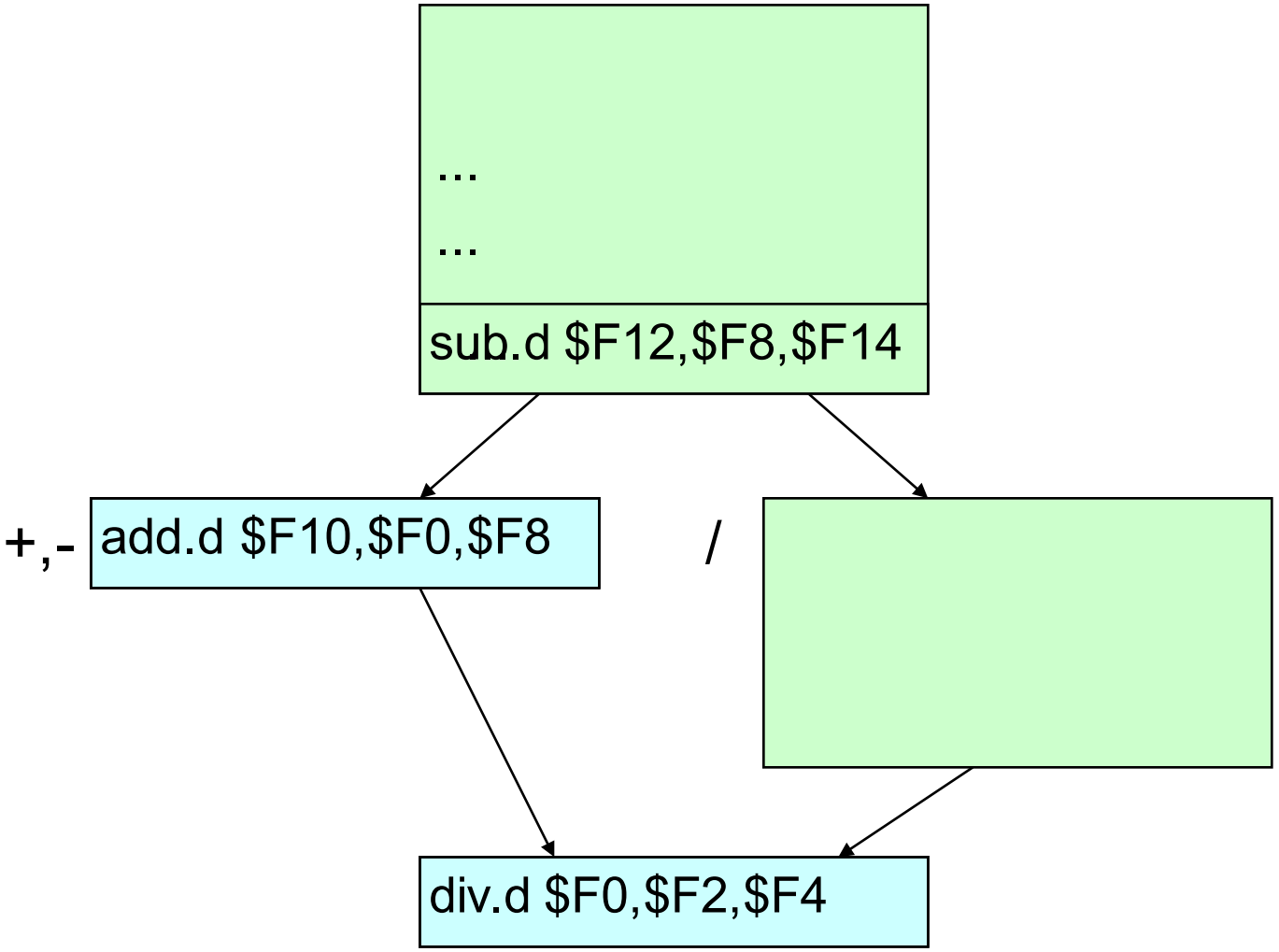
- in-order execution -



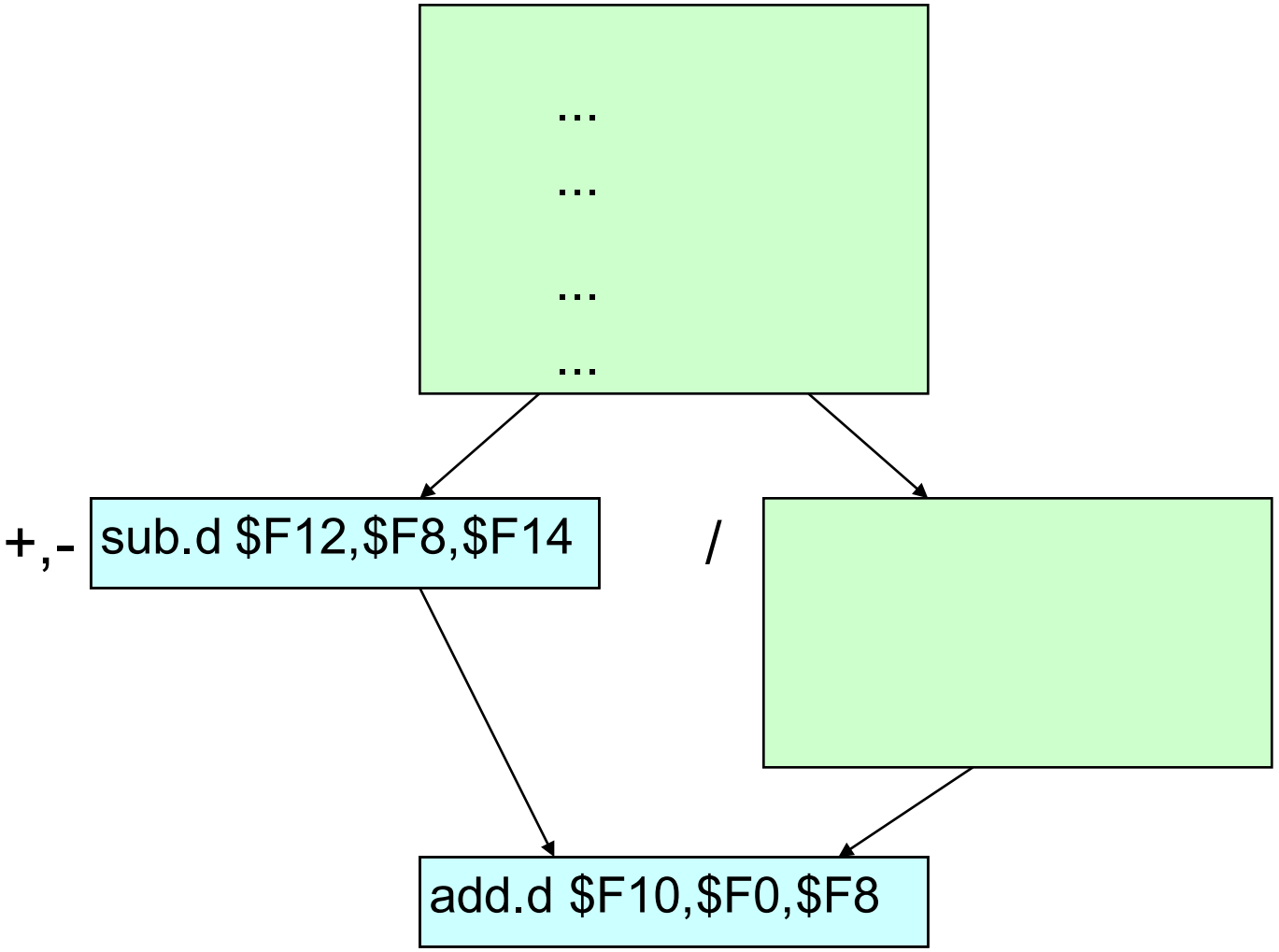
- in-order execution -



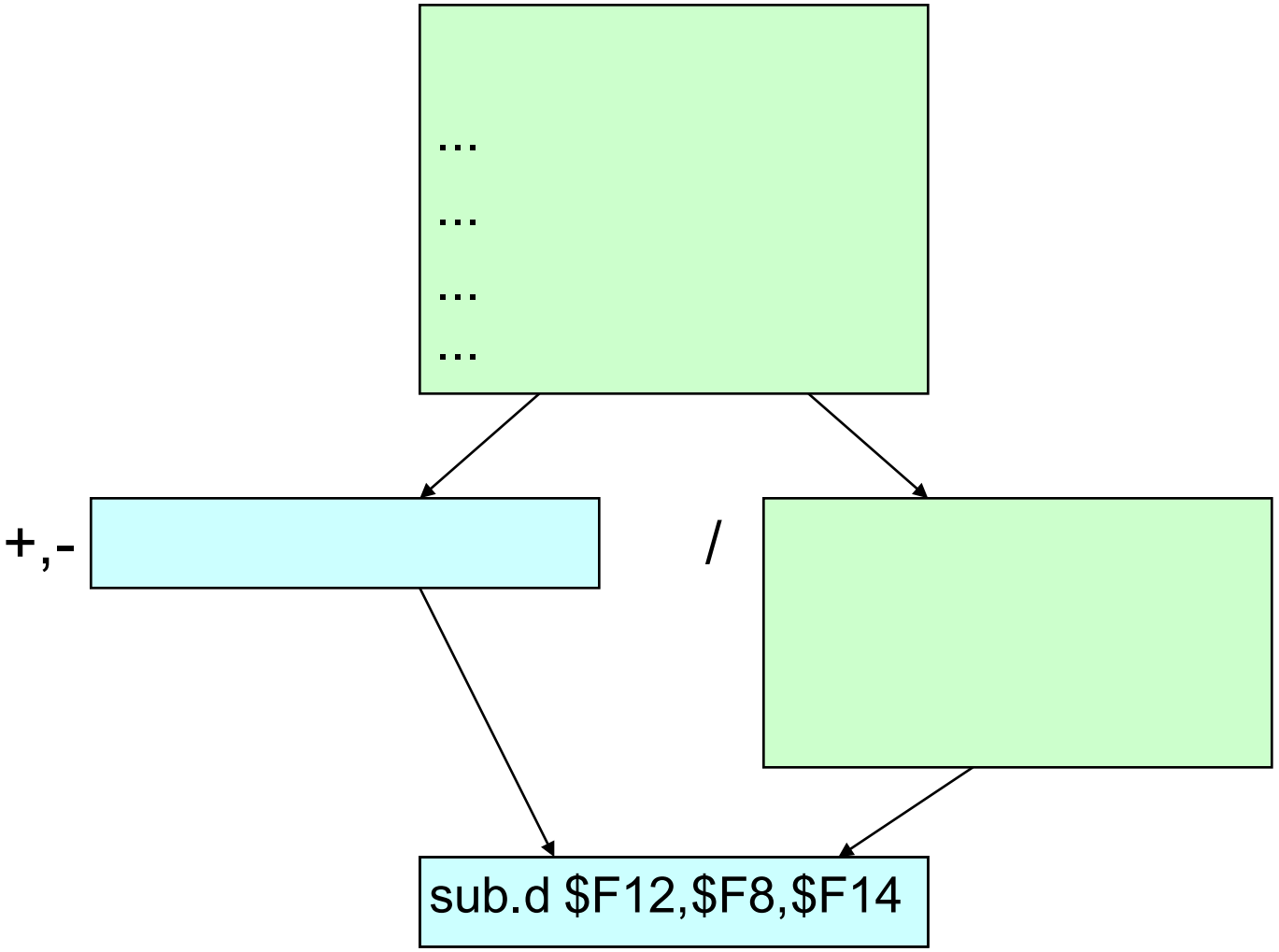
- in-order execution -



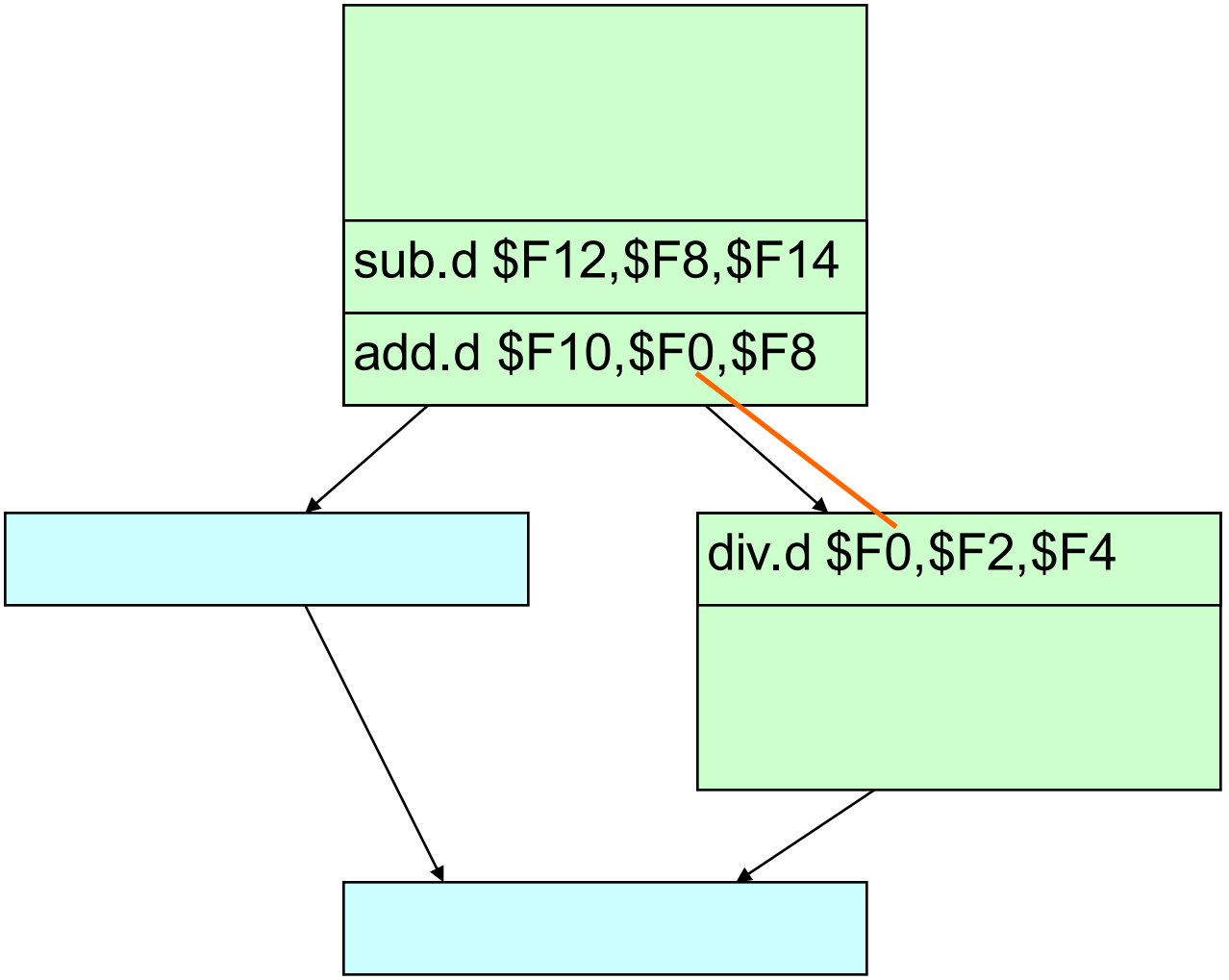
- in-order execution -



- in-order execution -

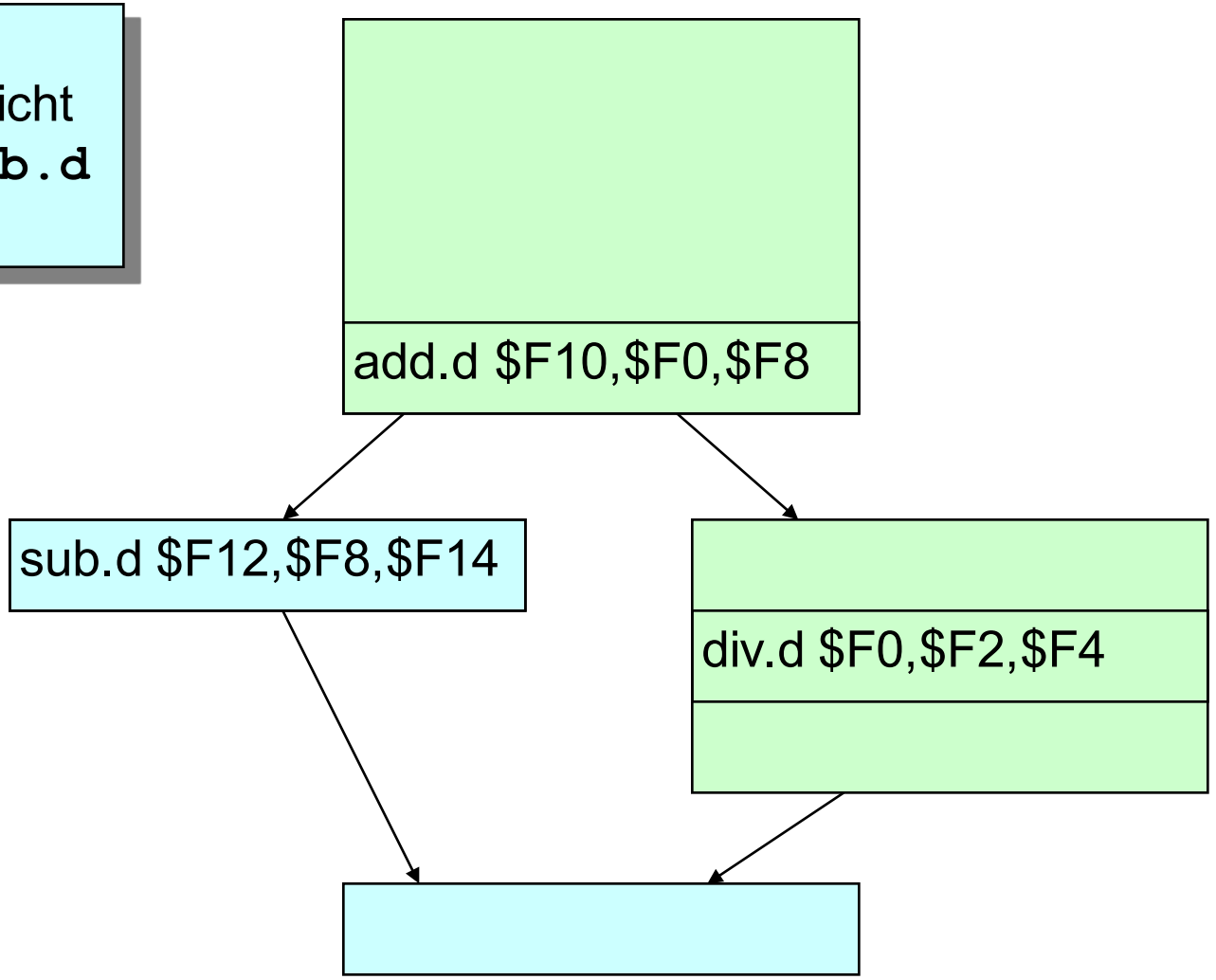


Dynamisches Scheduling von Befehlen
- out-of-order execution -



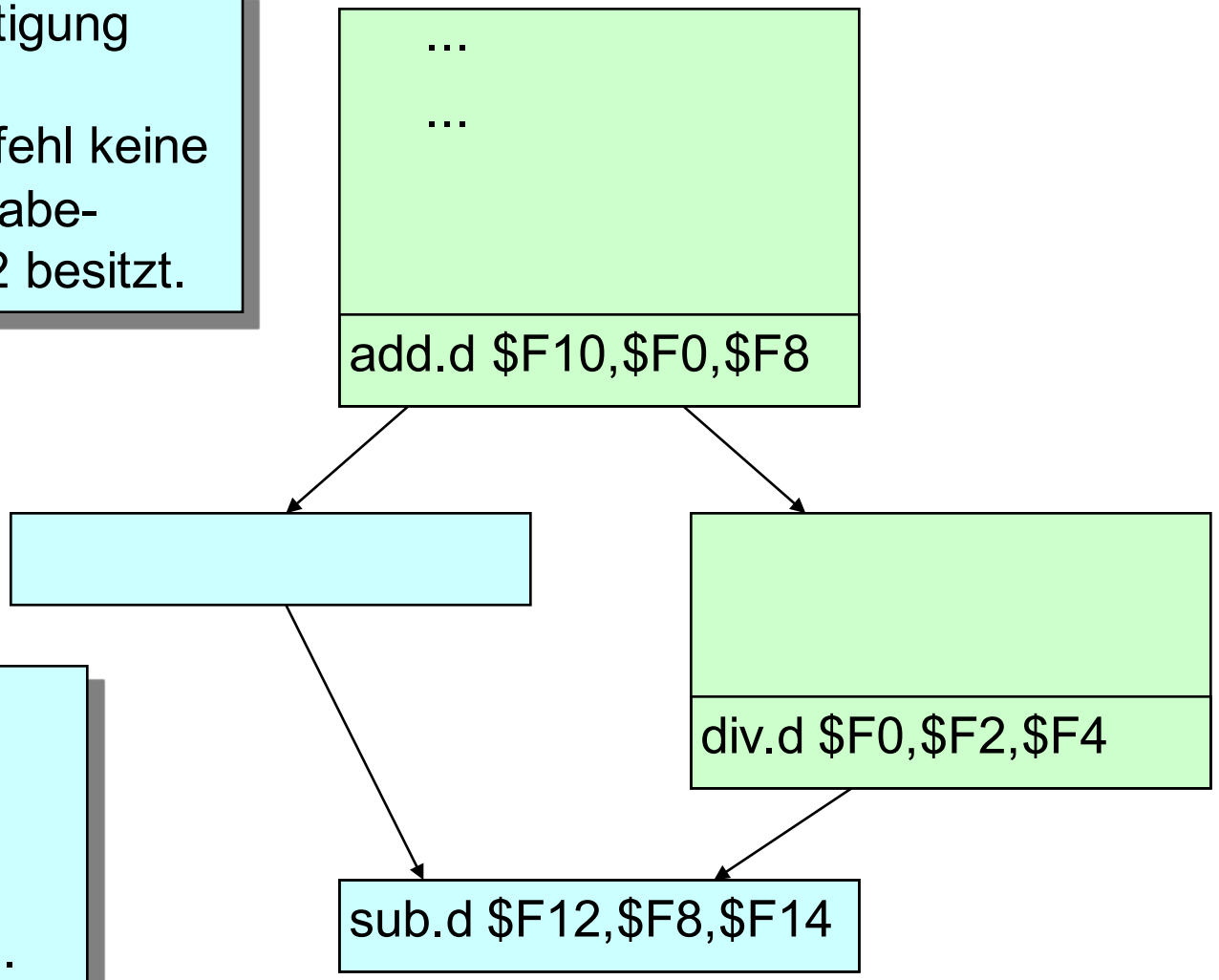
- out-of-order execution -

`add.d` kann wegen Datenabhängigkeit nicht gestartet werden, **`sub.d`** wird vorgezogen.



- out-of-order execution -

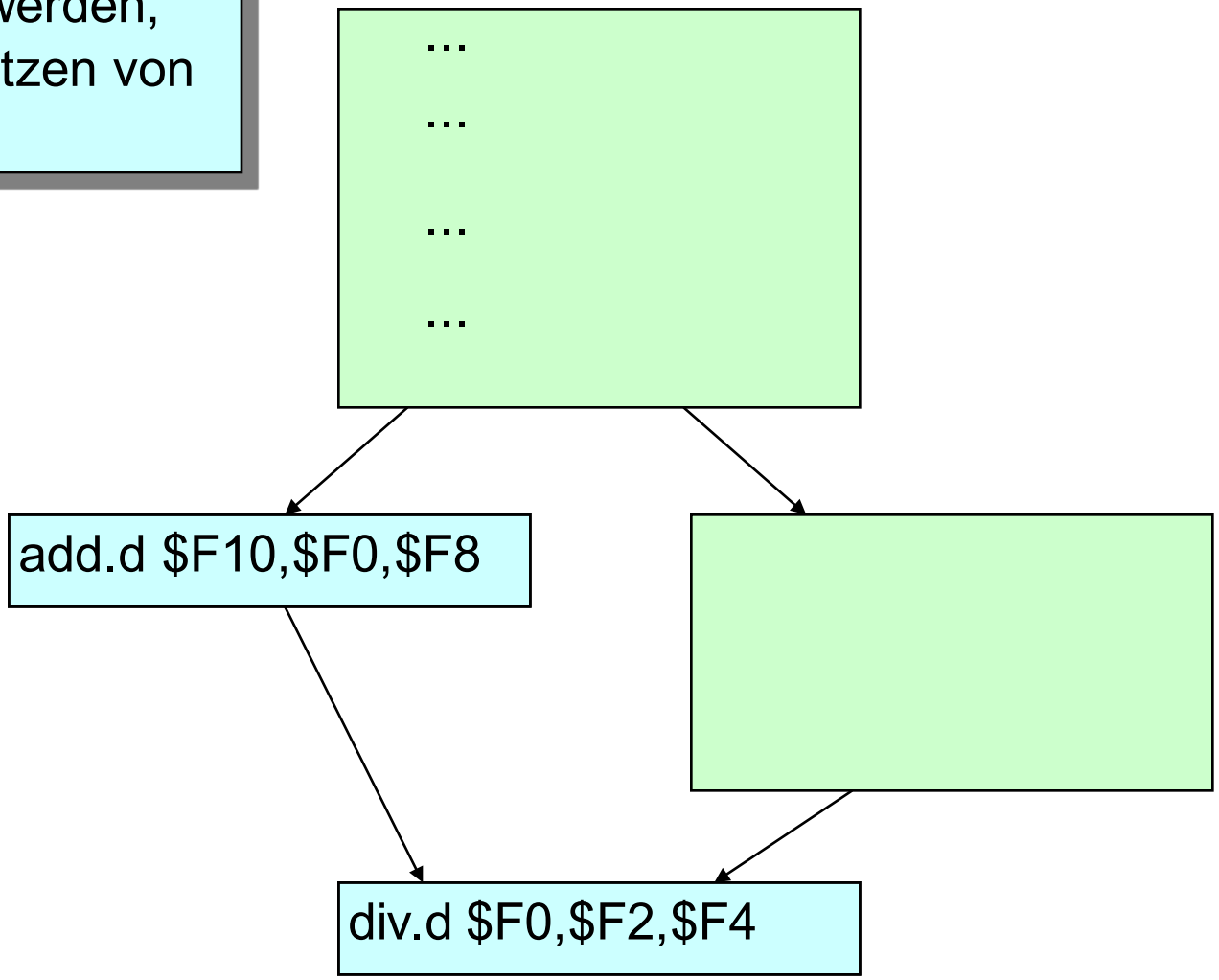
`sub.d` erhält Berechtigung zum Abspeichern, da wartender **`add.d`** Befehl keine Antidaten- oder Ausgabe-abhängigkeit zu $F12 besitzt.



Bei Antidaten- und Ausgabeabhängig-keiten keine Möglichkeit, andere Befehle zu überholen.

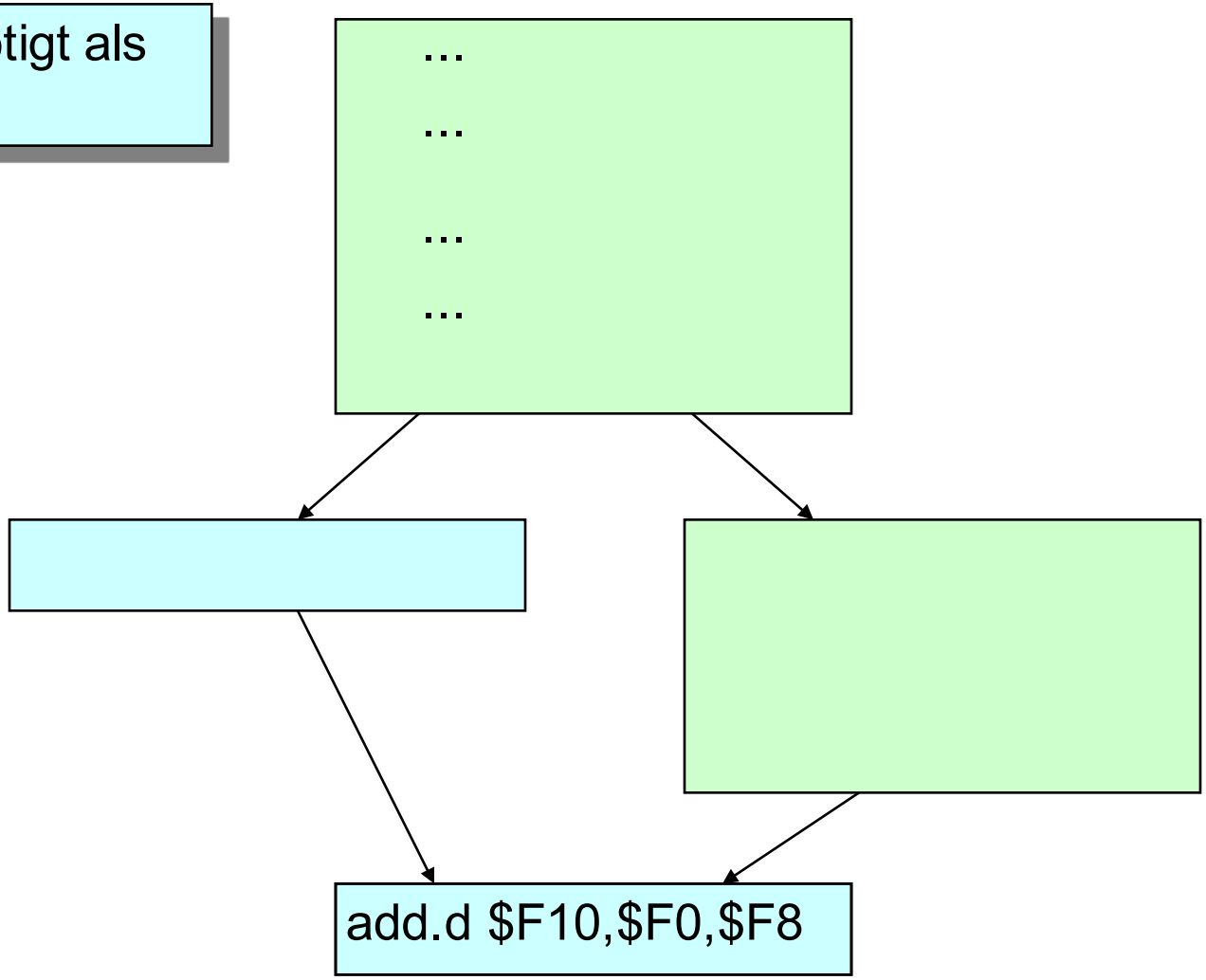
- out-of-order execution -

`add` kann gestartet werden, da Bypassing das Nutzen von $F0 gestattet.



- out-of-order execution -

1 Zyklus weniger benötigt als bei *in-order execution*.



3.3.2 *Scoreboarding*

- Jeder Befehl, der aus der *Instruction fetch*-Einheit kommt, durchläuft das *Scoreboard*.
- Wenn für einen Befehl alle Daten bekannt sind und die Ausführungseinheit frei ist, dann wird er gestartet.
- Alle Ausführungseinheiten melden abgeschlossene Berechnungen dem *Scoreboard*.
- Dieses erteilt Berechtigung zum Abspeichern, sofern
  - Speichereinheit frei ist und
  - die Antidaten- und die Ausgabeabhängigkeit berücksichtigt sind
- und prüft, ob dadurch Befehle ausführbereit werden.

Scoreboarding (2)

Zentrale Datenstruktur hierfür: *Scoreboard*
(deutsch etwa „Anzeigetafel" [für Befehlsstatus])

Ursprünglich war es realisiert für CDC 6600 (Jahr 1966):

- *load/store*-Architektur
- mehrere funktionelle Einheiten (4xFP, 6xMem, 7xInteger)

Scoreboarding für MIPS-Architektur ist nur sinnvoll

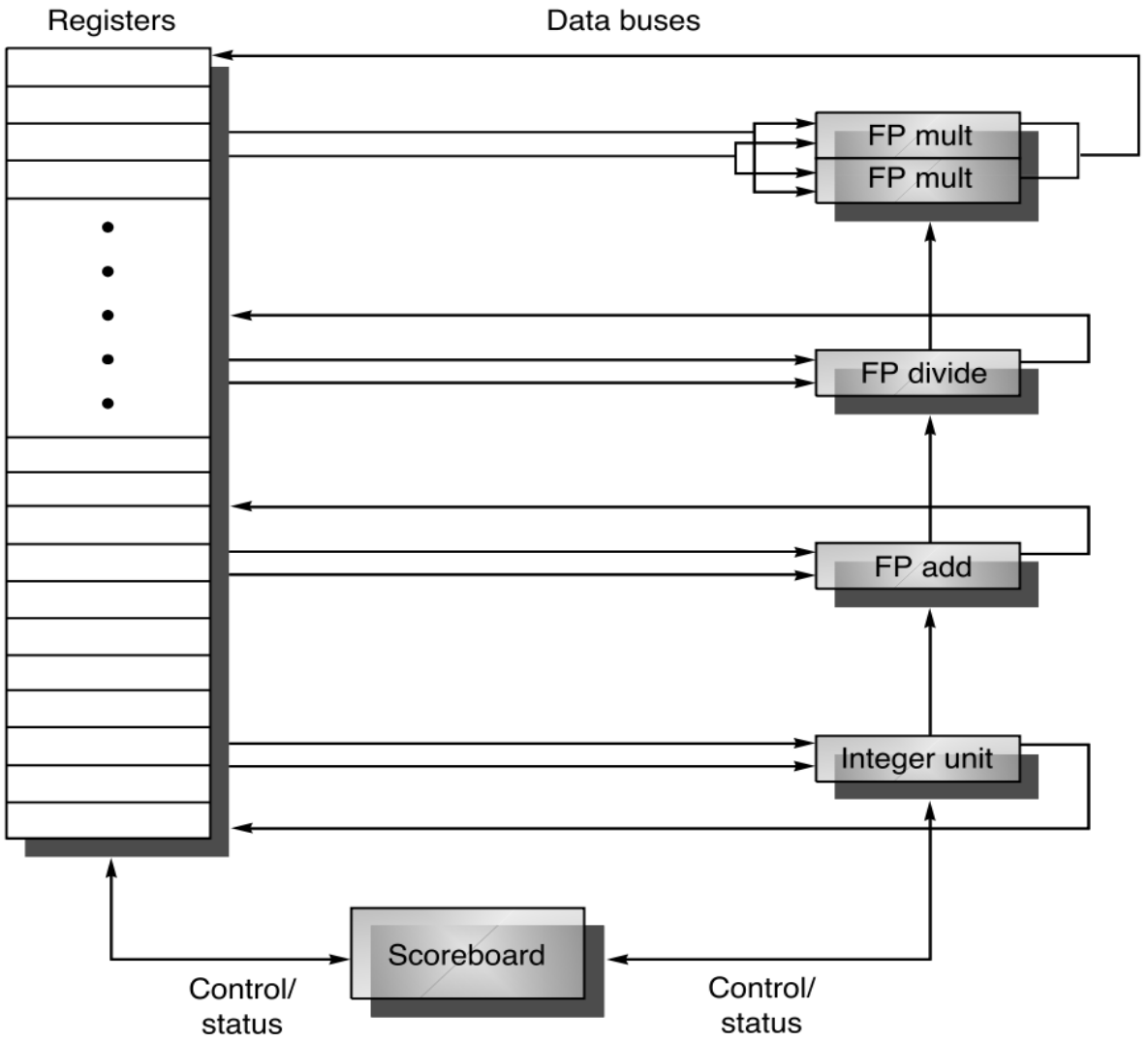
- für FP-Pipeline (Operationen mit mehreren Taktzyklen)
- und mehrere funktionelle Einheiten
  (Hier: 2 x Mult +Div + Add + Int).

Scoreboarding (3)

Struktur einer MIPS-*Pipeline* mit *Scoreboard*

Scoreboard kontrolliert funktionelle Einheiten

Bussysteme erforderlich



Funktion des *Scoreboards*

Scoreboard kontrolliert Befehlsausführung ab Befehlsholphase (IF) in 4 Phasen:

1. *Issue*
   - Befehl wird ausgegeben falls
     - benötigte funktionale Einheit ist frei und
     - kein anderer aktiver Befehl (d.h. im *Scoreboard*) schreibt gleiches Zielregister ☞ verhindert WAW-Konflikte
   - Auswertung beginnt noch nicht, Befehl nur an funktionelle Einheit!
     - Ersetzt Teil der ID-Phase in klassischer 5-stufigen Pipeline
     - Anhalten der *Pipeline* (*stall*) erfolgt:
       - Bei strukturellen Konflikten oder
       - WAW-Konflikt
       - ☞ kein weiterer Befehl wird ausgegeben! (warum?)

Funktion des *Scoreboards* (2)

2. *Read Operands* (RO)
   - *Scoreboard* beobachtet Verfügbarkeit von Quelloperanden
     ☞ sind verfügbar, wenn kein früher ausgegebener Befehl, der noch aktiv ist, diese schreiben wird (Datenflussprinzip)
   - Sobald Operanden verfügbar: *Scoreboard* signalisiert betreffender fkt. Einheit Beginn der Ausführungsphase
   - RAW-Konflikte (d.h. echte Datenabhängigkeiten) werden hier dynamisch aufgelöst
   - Befehle werden ggf. *out-of-order* zur Ausführung gebracht (*Issue* erfolgt noch strikt *in-order*!)
   - *Scoreboarding* verwendet kein *Forwarding*
     ☞ Operanden werden bei Verfügbarkeit aus Register gelesen

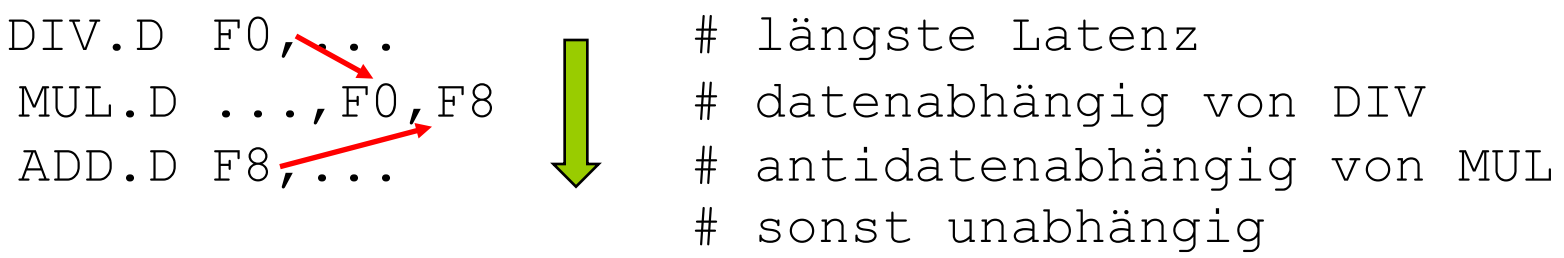
Funktion des Scoreboards (3)

3. *Execution*
   - Funktionelle Einheit beginnt Befehlsausführung nach Erhalt der Operanden
   - *Scoreboard* wird informiert, sobald Ergebnis vorliegt (noch in funktioneller Einheit, d.h. noch nicht im Register!)
   - Entspricht „klassischer" EX-Phase, ggf. >1 Zyklen lang
4. *Write Results* (WR)
   - Ausführungseinheit meldet abgeschlossene Berechnung an *Scoreboard*
   - *Scoreboard* erteilt Berechtigung zum Abspeichern, wenn Zielregister nicht von aktiver Operation (in RO) noch benötigt ☞ Lösung von WAR-Konflikten

Abspeichern macht ggf. Befehl ausführungsbereit

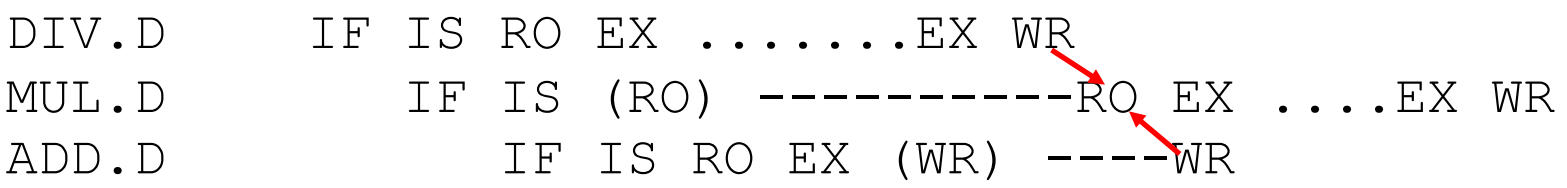
WR und RO dürfen nicht überlappen!

Beispiel



Annahme für Latenzen: DIV > MUL > ADD

Timing-Schema bei Scoreboarding:



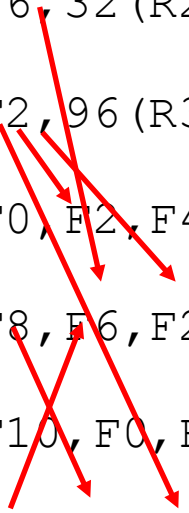
- Alle Befehle in-order ausgegeben (issue)
- Reihenfolgeänderung bei Ausführung via *Read Operands*
- WAR-Konflikt durch *stall* in *Write Results* aufgelöst

Datenstrukturen

1. **Befehlsstatus**: Phase (*Issue, ... Write Results*), in der sich Befehl befindet
2. **Status der funktionellen Einheiten**, unterteilt in:
   - *Busy* : Einheit arbeitend oder nicht
   - Op : Aktuelle Operation
   - Fi : Zielregister
   - Fj, Fk : Quellregister
   - Qj, Qk : fkt. Einheiten, die ggf. Quellregisterinhalt erzeugen
   - Rj, Rk : *Flag*, ob Quelloperanden verfügbar aber, noch nicht in *Read Operands* gelesen
3. **Register-Ergebnis-Status**
   Welche funktionelle Einheit verfügbare Register schreibt (ggf. leer)

Codebeispiel

```
L.D     F6,32(R2)

L.D     F2,96(R3)

MUL.D   F0,F2,F4      # datenabhängig vom 2. L.D

SUB.D   F8,F6,F2      # datenabh. vom 1. L.D und 2. L.D

DIV.D   F10,F0,F6     # datenabh. von MUL und 1. L.D

ADD.D   F6,F8,F2      # datenabh. von 2. L.D und
                      # antidatenabh. von SUB
```



Betrachten Situation im Scoreboard:

- Beide Loads (F2, F6) fertig bearbeitet
- Weitere Befehle soweit möglich in Bearbeitung fortgeschritten
- Letzter Schritt: Ergebnis von 1. Load (F6) gespeichert

Codebeispiel (2)

Befehls-Status

Befehl		Issue	Read Op.	Exec. complete	Write Res.
L.D	F6,32(R2)	+	+	+	+
L.D	F2,96(R3)	+	+	+	
MUL.D	F0,F2,F4	+			
SUB.D	F8,F6,F2	+			
DIV.D	F10,F0,F6	+			
ADD.D	F6,F8,F2				

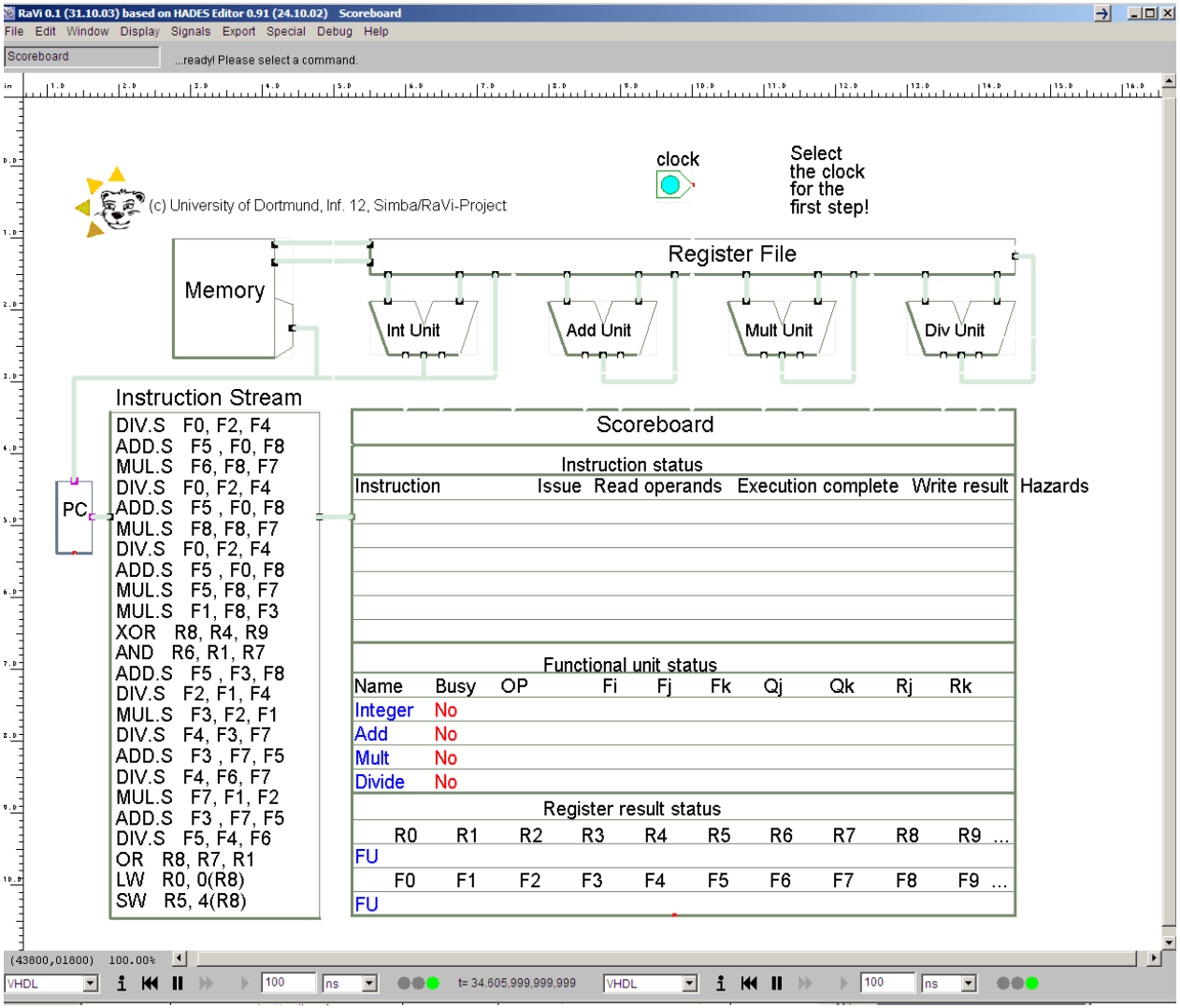
Status der Funktionseinheiten

Name	*Busy*	*Op*	*Fi*	*Fj*	*Fk*	*Qj*	*Qk*	*Rj*	*Rk*
Integer	ja	Load	F2	R3				nein	
Mult1	ja	Mult	F0	F2	F4	Int.		nein	ja
Mult2	nein								
Add	ja	Sub	F8	F6	F2		Int.	ja	nein
Divide	ja	Div	F10	F0	F6	Mult1		nein	ja

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*...*
Einheit	Mult1	Int			Add	Div	

Scoreboard-Simulation



http://ls12-www.cs.tu-dortmund.de/daes/de/lehre/downloads/ravi/download-and-installation.html

(Der Simulator ist nicht richtig um die strukturellen Konflikten zu löschen)

Bewertung

Kosten/Nutzen von *Scoreboarding* (für CDC 6600)

- Leistungssteigerung
  - um Faktor 1.7 (FORTRAN-Programme) bzw.
  - um Faktor 2.5 (handcodierter Assembler)
  - Bewertung vor neueren RISC-Entwicklungen bzw. optimierten Speicherhierarchien!
- Hardwareaufwand
  - *Scoreboard* umfasst Logik vergleichbar einer der funktionellen Einheiten (d.h. relativ wenig)
  - Hauptaufwand durch Bussysteme zwischen funktionellen Einheiten und Registern, bzw. zum *Scoreboard* (4x höher gegenüber Prozessor mit sequentieller Dekodierung)

Bewertung (2)

Nützt verfügbaren (!) ILP aus, um *Stalls* aufgrund von echten Datenabhängigkeiten zu reduzieren

Limitierungen entstehen durch:

- Umfang der Parallelität auf Instruktionsebene (ILP)
  (nur innerhalb von Basisblöcken betrachtbar, ggf. gering)
- Anzahl der Einträge im *Scoreboard*
  - Bestimmt, wie weit voraus die *Pipeline* nach unabhängigen Befehlen suchen kann
- Anzahl und Typ der funktionellen Einheiten
  - *Stall* bei strukturellen Konflikten hält alle weiteren Befehle an!
- Entstehen von WAR / WAW-Konflikten durch *Scoreboarding* ☞ erzeugen *stalls*, die Vorziehen der Befehlsbearbeitung wieder zunichte machen

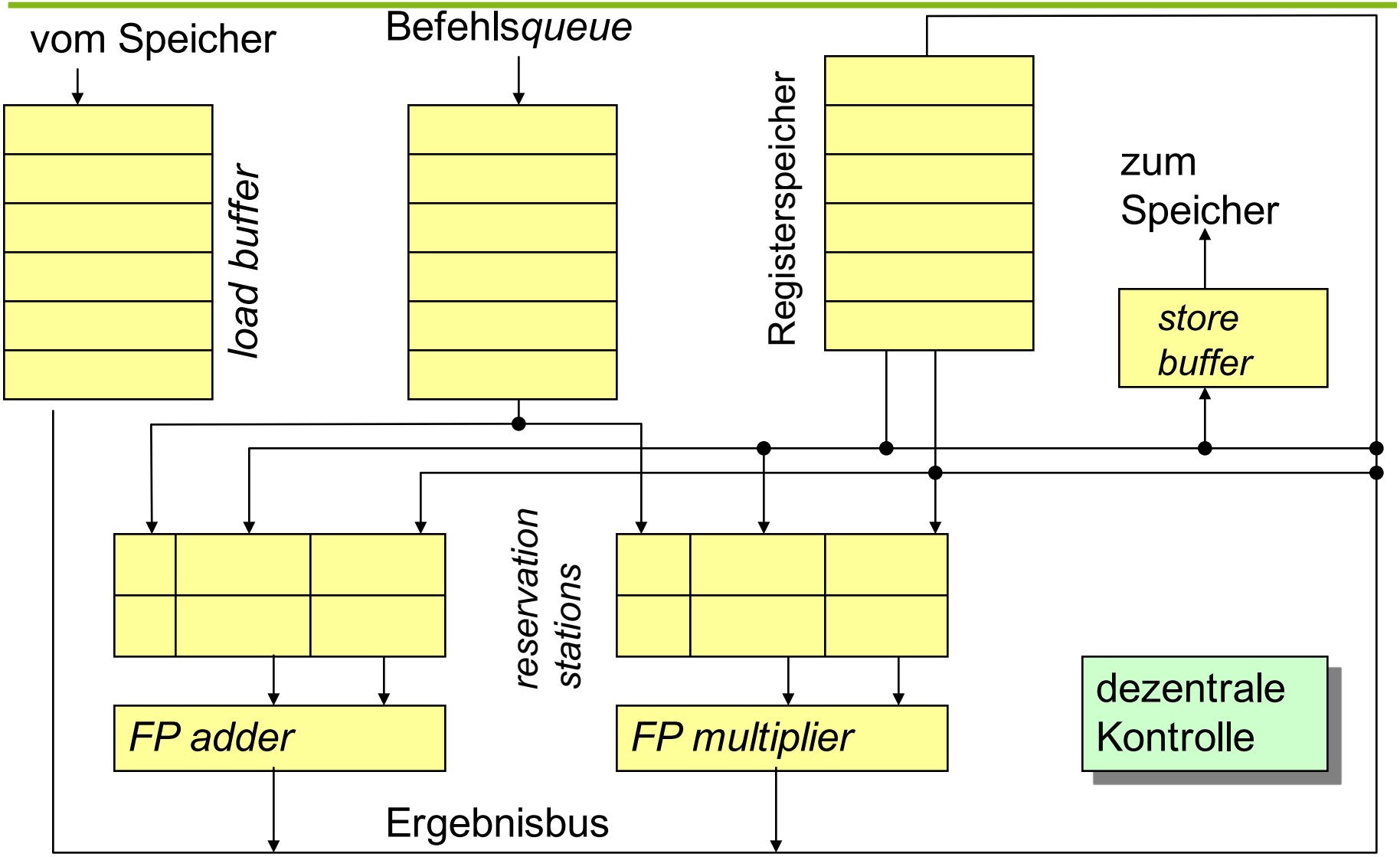
3.3.3 Verfahren von Tomasulo (1)

- Erdacht für IBM 360.
- Verfahren von Tomasulo erlaubt, auch bei Ausgabe- und Antidatenabhängigkeiten die Reihenfolge zu vertauschen.
- Umbenennung der Register, verschiedenen Benutzungen eines Registers werden verschiedene Speicherzellen zugeordnet.
- Jeder funktionellen Einheit wird eine ***Reservation Station*** zugeordnet.
- *Reservation stations* enthalten die auszuführende Operation und, soweit bekannt, die Operanden bzw. eine **Kennzeichnung** in Form von ***tag bits*** des Operanden.

Verfahren von Tomasulo (2)

- Sind alle Operanden bekannt und ist die funktionelle Einheit frei, so kann die Bearbeitung beginnen.
- Am Ende der Bearbeitung wird das Ergebnis von allen Einheiten übernommen, die das Ergebnis benötigen.
- Verteilen der Daten erfolgt vor der Abspeicherung im Registerspeicher.
- Aus den *tag bits* geht hervor, aus welcher Einheit der Operand kommen muss.
- Registeradressen werden dynamisch auf größere Anzahl von Plätzen in den *reservation stations* abgebildet, Register effektiv umbenannt. *Performance*-Beschränkungen wegen weniger Register umgangen.

Verfahren von Tomasulo (3)



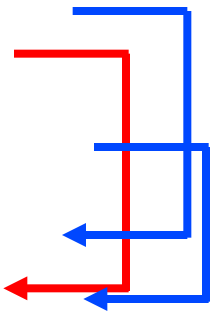
Ursprüngliche Motivation/Kontext

- Hohe FP-Leistung von Befehlssatz und Compiler für gesamte IBM 360-Familie (d.h. nicht durch Spezialisierung)
- Überwindung von Einschränkungen durch nur 4 architekturell definierte FP-Register (*Compiler Scheduling* kaum möglich)
- Lange Latenzen bei Speicherzugriffen und FP-Operationen
- Benutzt erstmalig (implizite) Register-Umbenennung
  ☞ Minimierung von WAR und WAW-Konflikten

Register Renaming

Beispielcode mit Namensabhängigkeiten:

```
DIV.D   F0,F2,F4
ADD.D   F6,F0,F8        # Lesen von F8,
                        # Schreiben von F6
S.D     F6,0(R1)        # Lesen von F6
SUB.D   F8,F10,F14      # Schreiben von F8
MUL.D   F6,F10,F8       # Schreiben von F6
```



- Antidatenabhängigkeit zwischen ADD.D und SUB.D (via F8) sowie zwischen S.D und MUL.D (via F6)
- Ausgabeabhängigkeit zwischen ADD.D und MUL.D (via F6)
- Es existieren auch **echte** Datenabhängigkeiten

Register Renaming (2)

Prinzip: Wir verwenden temporärer Register für (logisch) neue, möglicherweise interferierende Belegung

Beispiel (cont)

- Annahme: zwei temporäre Register S und T
- Neue Belegungen von F8 durch SUB bzw. F6 durch MUL in (eindeutige) temporäre Register „umleiten”

```
DIV.D   F0,F2,F4
ADD.D   F6,F0,F8        # Lesen von F8,
                        # Schreiben von F6
S.D     F6,0(R1)        # Lesen von F6
SUB.D   S,F10,F14       # Schreiben von S (F8)
MUL.D   T,F10,S         # Schreiben von T (F6)
```

- Alle Namenskonflikte durch Umbenennung auflösbar (Voraussetzung: Genügend temporäre Register)
- Weitere Verwendung von F6/F8 durch S/T ersetzen!

Verfahren von Tomasulo: Prinzip

Wichtige Hardwarestruktur: *Reservation Stations*

- Zugeordnet zu funktionalen Einheiten (i.d.R. je eine)

Arbeitsweise von *Reservation Stations*:

- Puffern Operanden für Befehle (sobald verfügbar geladen)
  ☞ Müssen nicht aus Registern gelesen werden!
- Ausstehende Operanden verweisen auf *Reservation Station*, die Eingabe bereitstellen wird
- Bei aufeinander folgenden Schreibzugriffen auf Register:
  Nur letzter für Aktualisierung des Inhalts verwendet

Falls |*Reservation Stations*| > |Register| ☞ Namenskonflikte sind lösbar, die Compiler nicht behandeln kann!

Verfahren von Tomasulo: Prinzip (2)

Verwendung von *reservation stations* anstelle des zentralen Registersatzes bewirkt 2 wichtige Eigenschaften:

- Konfliktdetektion und Ausführungskontrolle verteilt
  - Informationen in *reservation stations* bei den funktionellen Einheiten bestimmen, wann Ausführung eines Befehls möglich
- Ergebnisse werden direkt zu den funktionellen Einheiten (in jeweiliger *reservation station*) weitergereicht
  - Erweiterte Form des *Forwarding*
  - Realisiert implizit *Register Renaming*
  - Möglich durch gemeinsamen Ergebnisbus (*common data bus*)
    ☞ mehrfach vorhanden, falls >1 Ergebnis/Takt weiterzuleiten

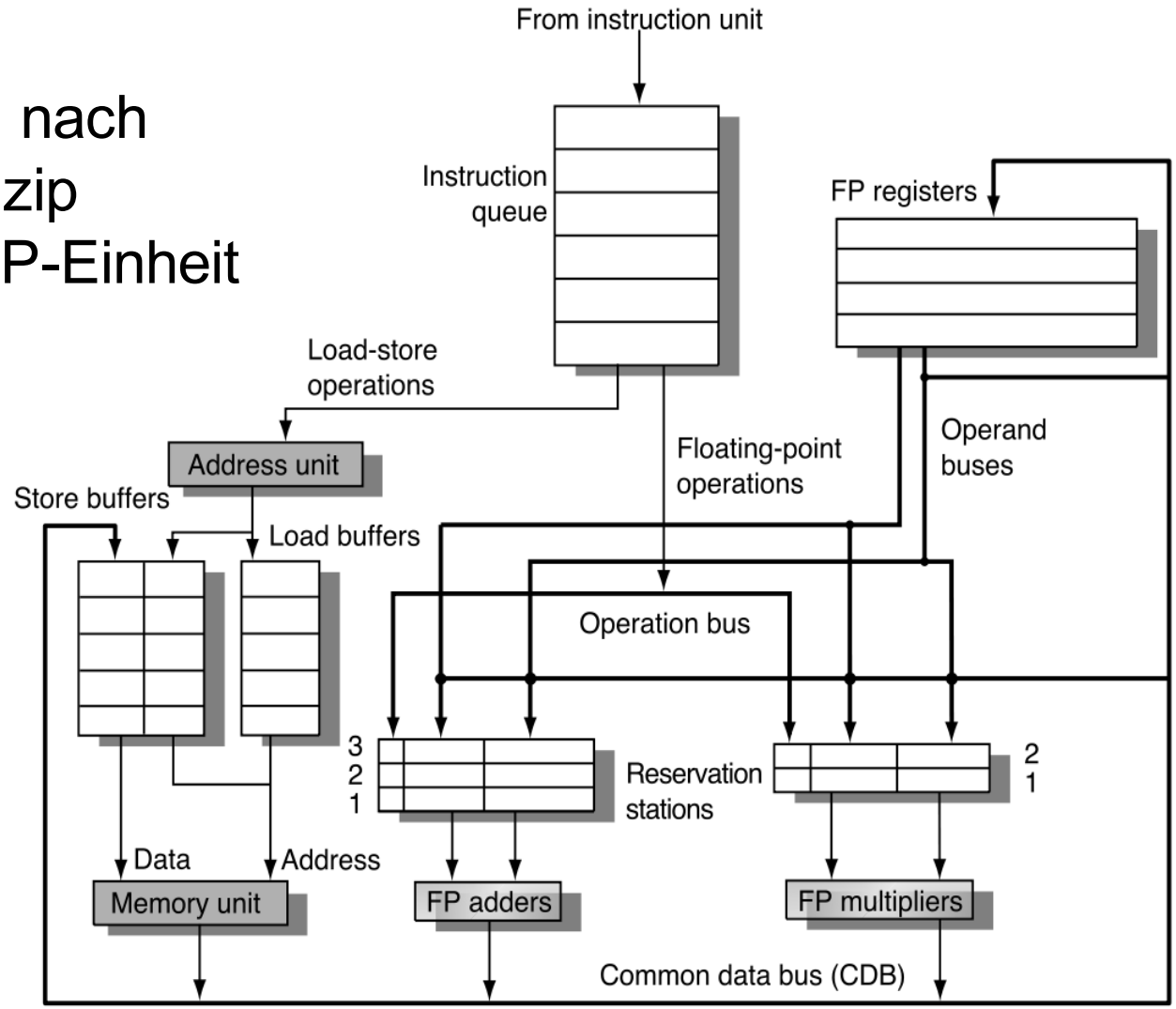
Verfahren von Tomasulo: Prinzip (3)

Grundstruktur einer nach dem Tomasulo-Prinzip realisierten MIPS-FP-Einheit

Nur realisiert:

- FP-Ops. und
- Load/Store

Load/Store Buffer vergleichbar *reservation stations*



Bearbeitungsphasen (1)

Ausführung eines Befehls in 3 Bearbeitungsphasen:

1. *Issue*
   - Befehl aus Befehlswarteschlange lesen (IF bereits erfolgt)
   - In *Reservation Station* der benötigten funktionellen Einheit eintragen, sofern eine frei
   - Operandenwerte aus Registern lesen (sofern dort vorhanden)
   - Falls Operanden erst erzeugt werden müssen, Referenz auf erzeugende Reservation Station eintragen (und Entstehen „beobachten")
   - *Reservation Stations* wie weitere Register behandelt
     ☞ WAR und WAW-Konflikte auflösbar
     (*Register Renaming* damit implizit erreicht)

Bearbeitungsphasen (2)

2. *Execution*
   - *reservation stations* beobachten Ergebnis-Bus, sofern Operanden noch nicht verfügbar
   - Neu berechnete Operanden in *reservation stations* eintragen
   - Sobald alle Operanden verfügbar: Abarbeitung des Befehls in korrespondierender funkt. Einheit starten
   - Hinweise:
     - RAW-Konflikte durch datengetriebenes Starten der Befehlsausführung gelöst („Warten" = *Pipeline-Stalls*)
     - > 1 Befehle können gleichzeitig ausführbereit werden
3. *Write Results*
   - Ergebnisse werden via Ergebnis-Bus in wartende *reservation stations* und ggf. in Register geschrieben

Verfahren von Tomasulo: Datenstrukturen

Notation in Anlehnung an *Scoreboarding*

- 1. Befehlsstatus: Phase (*Issue, ... Write Results*) in der sich Befehl befindet
- 2. Status der *reservation stations*:
  - *Busy*: Einheit arbeitend oder nicht
  - Op: Aktuelle Operation
  - Qj, Qk: *reservation stations*, die ggf. Quelloperanden erzeugen (leer, wenn bereits in Vj/Vk verfügbar)
  - Vj, Vk: Werte der Quelloperanden
- Für *Load* bzw. *Store*-Puffer:
  - A: Effektive Adresse für Speichertransfer
- 3. Register-Ergebnis-Status
  - Qi: *reservation station*, die Operation steuert, deren Ergebnis in aktuellem Register gespeichert werden soll

Datenstrukturen (2)

Prozessorkonfiguration:

1 Adress- und 1 Speichereinheit, je 5 Load u. Store-Puffer

1 Addiereinheit, 3 Puffer (in Reservation Stations), und 1 Multiplizierereinheiten, 2 Puffer in Reservation Stations

Codebeispiel

```
L.D      F6,32(R2)
L.D      F2,96(R3)
MUL.D    F0,F2,F4        # datenabhängig vom 2. L.D
SUB.D    F8,F6,F2        # datenabh. vom 1. LD und 2. LD
DIV.D    F10,F0,F6       # datenabh. von MUL und 1. LD
ADD.D    F6,F8,F2        # datenabh. von 2. LD und
                         # antidatenabh. Von 1. SUB
SUB.D    F8,F4,F0        # ausgabeabhängig von 1. SUB
```

Betrachten Situation / Zustand der *reservation stations*:

- 1. Load (F6) fertig bearbeitet, Ergebnis geschrieben
- 2. Load (F2) eff. Adresse berechnet, wartet auf Speicher
- Andere Befehle soweit möglich ausgegeben (nicht 2. SUB)

Datenstrukturen (3)

Befehls-Status

Befehl		*Issue*	*Execute*	*Write Result*
L.D	F6,32(R2)	+	+	+
L.D	F2,96(R3)	+	+	
MUL.D	F0,F2,F4	+		
SUB.D	F8,F2,F6	+		
DIV.D	F10,F0,F6	+		
ADD.D	F6,F8,F2	+		

Status der *reservation stations*

Name	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*	*A*
Load1	nein						
Load2	ja	Load					96+Regs[R3]
Add1	ja	SUB		Mem[32+Regs[R2]]	Load2		
Add2	ja	ADD			Add1	Load2	
Add3	nein						
Mult1	ja	MUL		Regs[F4]	Load2		
Mult2	ja	DIV		Mem[32+Regs[R2]]	Mult1		

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*...*
Qi	Mult1	Load2		Add2	Add1	Mult2	

Datenstrukturen (4)

Befehls-Status

Befehl		*Issue*	*Execute*	*Write Result*
L.D	F6,32(R2)	+	+	+
L.D	F2,96(R3)	+	+	
MUL.D	F0,F2,F4	+		
SUB.D	F8,F2,F6	+		
DIV.D	F10,F0,F6	+		
ADD.D	F6,F8,F2	+		
SUB.D	F8,F4,F0	+		

Status der *reservation stations*

Name	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*	*A*
Load1	nein						
Load2	ja	Load					96+Regs[R3]
Add1	ja	SUB		Mem[32+Regs[R2]]	Load2		
Add2	ja	ADD			Add1	Load2	
Add3	*ja*	*SUB*		*Regs[F4]*	*Mult1*		
Mult1	ja	MUL		Regs[F4]	Load2		
Mult2	ja	DIV		Mem[32+Regs[R2]]	Mult1		

ADD.D bekommt Kopie von F8, letzter Befehl darf F8 überschreiben

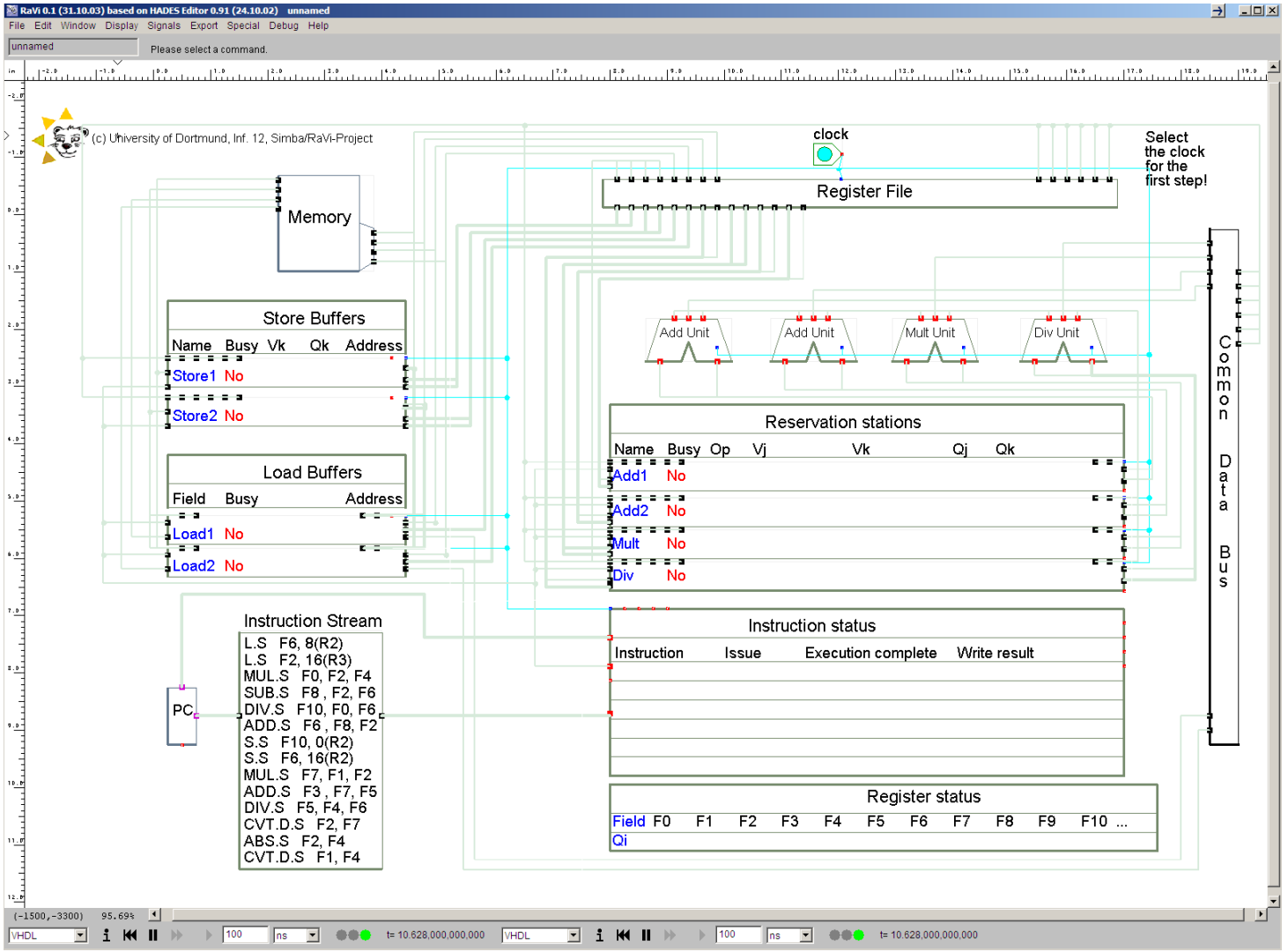
Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	...
Qi	Mult1	Load2		Add2	~~Add1~~ Add3	Mult2	

Auflösung der Ausgabeabhängigkeit (weitere “Verwendung” von F8 via Res.Stat.)

Tomasulo-Simulation

http://ls12-www.cs.tu-dortmund.de/daes/de/lehre/downloads/ravi/download-and-installation.html



Zusammenfassung

Dynamisches Scheduling

- Motivation für *out-of-order execution*
- *Scoreboarding*
- Verfahren von Tomasulo
  - Ein Form von Datenflussarchitektur innerhalb einer von-Neumann-Maschine
  - Umgekehrt kann das Tomasulo-Verfahren auf Taskebene benutzt werden.