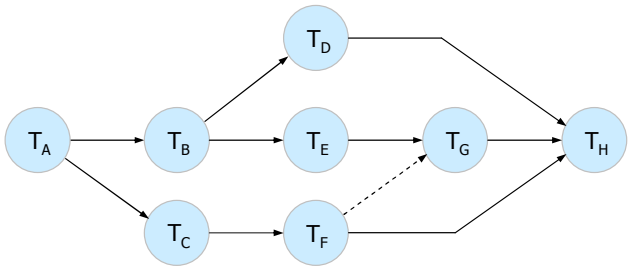
Task & Data-Flow Graphs

M. Thaler, TG208, tham@zhaw.ch
www.zhaw.ch/~tham

Um was geht es?

- **Parallele Algorithmen**
  - was sind Möglichkeiten zur Darstellung?
  - was sind Möglichkeiten zur Analyse?
  - welche Resultate lassen sich bestimmen?
- **Antwort: Task- resp. Datenfluss-Graphen**



- Funktionale Sprachen
  - lassen sich einfach auf Task Graphen abbilden

Lehrziele

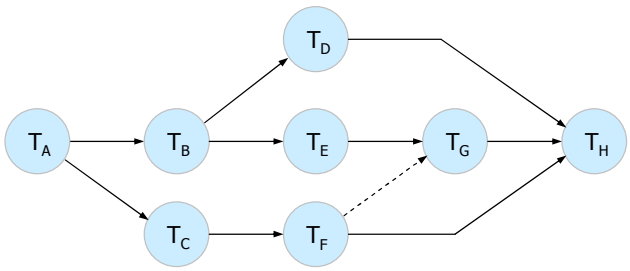
- **Sie können**
  - mit einem einfachen Beispiel erklären, was ein Task- resp. Datenflussgraph ist
  - erklären und diskutieren welche Abhängigkeiten zwischen Tasks existieren können
  - einen Task Graphen als Netzplan darstellen und die wichsten Eigenschaften bestimmen und erklären, was der kritische Pfad ist
  - Sie können einen Task Graphen als Balkendiagramm (Gantt Chart) zeichnen resp. interpretieren
  - anhand einfacher Beispiele zeigen, wie Ressourcen-Optimierungen mit Hilfe von Task Graphen gemacht werden können

Inhalt

- **Task und Data-Flow Graphs**
  - Darstellung
  - Abhängigkeiten
- **Analyse von Task Graphen**
  - "Netzplantechik"
  - Balkendiagramme
- **Fallbeispiel**

Task Graphen

- **Graphische Darstellung von Abhängigkeiten**
  - Tasks → Kreise
  - Datenabhängigkeit → Pfeile
  - Kontrollabhängigkeit → gestrichelte Pfeile

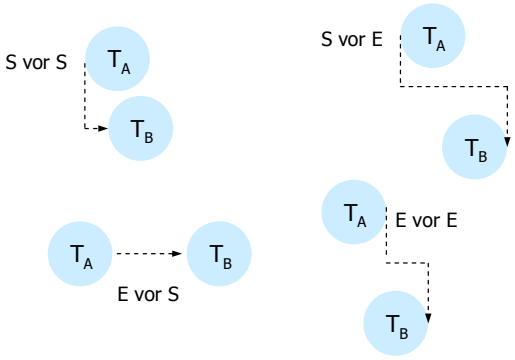


  - wenn nur Datenabhängigkeiten → Datenflussgraph

Abhängigkeiten

- **Abhängigkeiten nach**
  - **Daten** i.d.R.
    - **Ende → Start**

  - **Kontrolle**
    - Start → Start
    - Start → Ende
    - **Ende → Start**
    - Ende → Ende

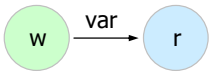


- Kontrollabhängigkeiten
  - Tasks müssen in einer bestimmten Reihenfolge abgearbeitet werden
  - Beispiel
    - Daten in zwei Teile partitioniert
    - je ein Task bearbeitet die Daten
    - nächster Schritt erst, wenn beide Tasks fertig

- mögliche Kontrollabhängigkeiten sind
  - S → S
    - Task T_A muss vor Task T_B starten
  - S → E
    - Task T_A muss starten, bevor T_B beendet ist
  - E → S
    - Task T_A muss beendet sein bevor Task T_B starten kann
  - E → E
    - Task T_A muss beendet sein, bevor Task T_B beedet ist

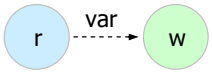
Read/Write Abhängigkeiten

- **Flow dependency**
  - write var → read var



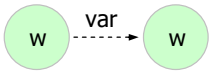

```
a = ...
b = a
```

- **Anti-dependency**
  - read var → write var




```
b = a
a = ...
```

- **Output-dependency**
  - write var → write var




```
a = ...
b = a
a = ...
```

- **Anti- / Output-dependence**
  - keine echten Datenabhängigkeiten

- Flow-dependencies
  - können nicht entfernt werden
- Anti-dependencies
  - können entfernt werden → Renaming
- Output-dependencies
  - können entfernt werden → Renaming

… Read/Write Abhängigkeiten

- **Beispiel**
  - sequentielles Programm

```
sum = a + 1;
on  = sum*s1;
sum = b + 3;
tw  = sum*s2;
```

flow dependency

anti-dependency

flow dependency

Renaming

```
sum0 = a + 1;
on   = sum0*s1;

sum1 = b + 3;
tw   = sum1*s2;
```

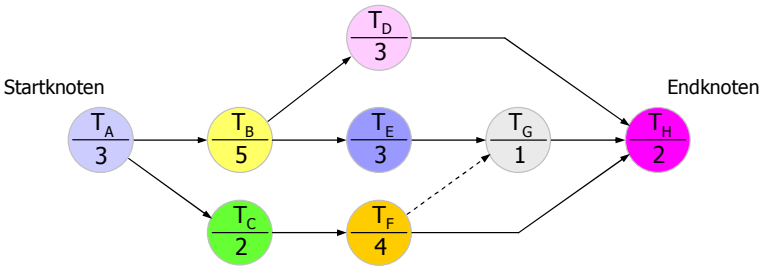
  - nach Renaming
    - Parallelverarbeitung möglich

- Flow dependencies
  - können nicht entfernt werden
- Anti-dependencies
  - können entfernt werden → renaming
- Output-dependencies
  - können entfernt werden → renaming

- Hinweis
  - anstelle von einfachen Operationen auch Funktionen möglich
  - Funktionale Sprachen (Lisp, etc.)

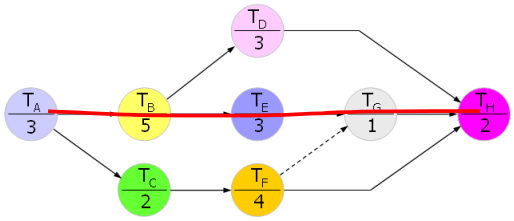
Analyse von Tasks Graphen

- **Tasks mit Ausführungszeiten**
  - jeder Task wird mit einer Ausführungszeit versehen



  - Ausführungszeiten pro Task
    - Messwerte: d, d_{min}, d_{max}, d_{mean}
    - Schätzwerte
    - etc.

... Task Graph



- **"Netzplantechnik"**
  - Berechnung von
    - t_E earliest start time
    - t_L latest start time (bei gleicher Endzeit)
    - slack = t_L - t_E

Task T_i	Dauer t_i	$t_E(i)$	$t_L(i)$	slack(i)
T_A	3	0	0	0
T_B	5	3	3	0
T_C	2	3	5	2
T_D	3	8	9	1
T_E	3	8	8	0
T_F	4	5	7	2
T_G	1	11	11	0
T_H	2	12	12	0

$T_1 = \sum d_i \quad \forall i = 23$

$T_\infty = \max(t_E(i) + d_i) = 14$

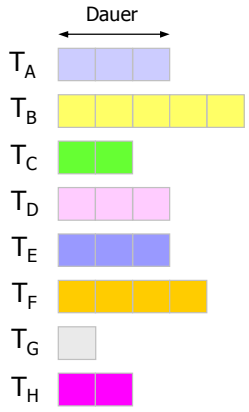
MPC ZHAW, MPC FS16, M. Thaler

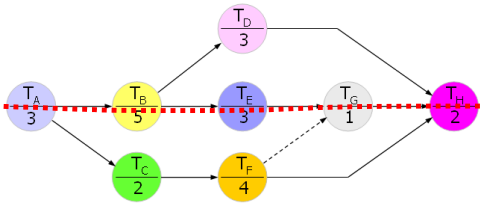
Februar 16 10

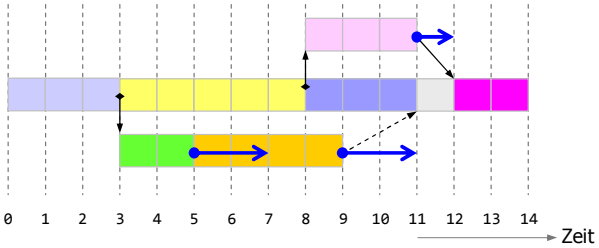
- Dauer eines Tasks: d_i
  - Berechnung, Messung, etc. (hier Vorgaben)
  - bei nicht konstanten d_i
    - maximum: t_{imax} → worst case
    - statistische Grössen PERT Netzplantechnik
- Berechnung von t_E und t_L
  - $t_E(i)$: t_E von Knoten i — frühest möglicher Startzeitpunkt
  - $t_L(i)$: t_L von Knoten i — spätest möglicher Startzeitpunkt
  - $t_E(i) = \max(t_E(j) + d_j)$ — j = alle Vorgänger von i
  - $t_L(i) = \min(t_L(k) - d_i)$ — k = alle Nachfolger von i
- Berechnung des slack (Pufferzeit)
  - slack = $t_L(i) - t_E(i)$
- Minimale Rechenzeit T_∞
  - "unendliche" Parallelität → minimal mögliche Ausführungsgzeit
- Maximale Rechenzeit T_1
  - keine Parallelität → maximale Ausführungszeit (Tasks seriell)
- **Kritischer Pfad:** alle Tasks T_i mit slack(i) = 0
  - nicht verschiebbar, ohne Gesamtdauer zu erhöhen
  - es gilt: $T_{min} = \sum d_i$, d_i auf kritischem Pfad
- Maximaler Speedup: $S_{max} = T_1 / T_\infty = \sum d_i / T_{min} = 23 / 14 = \sim 1.6$

... Task Graph

- **Balkendiagramm**
  - Länge ~ Dauer



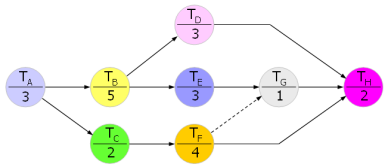




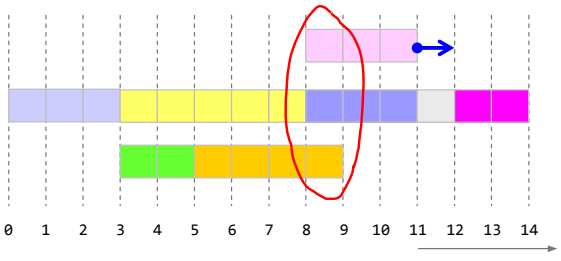
Balkendiagram / Gantt Chart

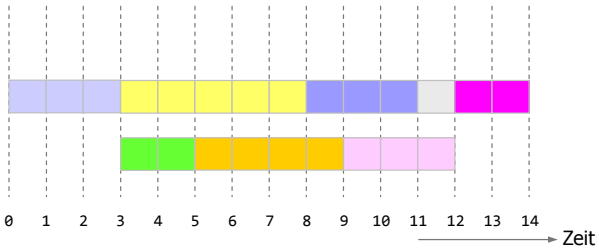
- Gant-Chart oder Balkendiagramm
  - Tasks als Balken mit Länge proportional zu Dauer einzeichnen
  - Abhängigkeiten aus DAG bestimmt Reihenfolge
  - Tasks so früh wie möglich einzeichnen

... Task Graph



- **Nutzen der Pufferzeit**
  - Ressourcenbedarf optimieren → "verschieben" von Tasks



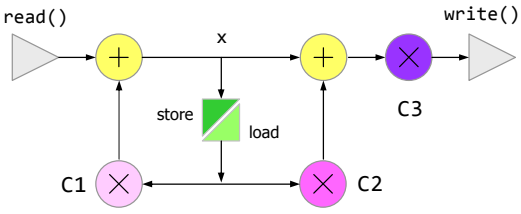


- Task T_D verschieben
  - verlängert Rechenzeit nicht
  - reduziert Ressourcenbedarf von 3 auf 2

Fallbeispiel

- **Digitalfilter 1. Ordnung**
  - Differenzengleichung und Datenflussdiagramm

$$x(n) = C_0 \cdot x(n-1) + u(n)$$
$$y(n) = C_3 \cdot \left(x(n) + C_2 \cdot x(n-1)\right)$$



  - C-Programm

```
while(1) {
  tmp  =  x * C1 + read();
  out  = (x * C2 + tmp) * C3;
  x = tmp;
  write(out);
}
```

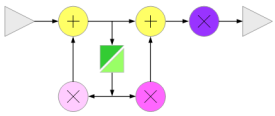
Folgendes Beispiel:
3 faches Loop Unrolling

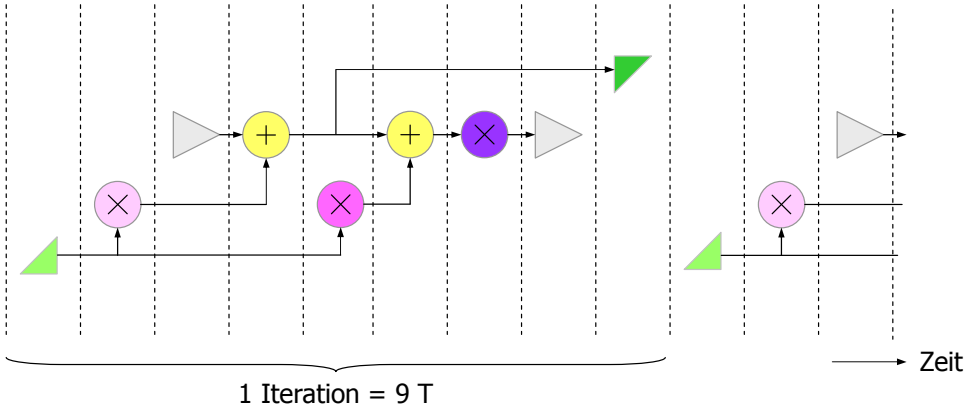
- Datenflussgraphen lassen sich auf verschiedensten Hierarchie-Ebenen darstellen
  - hier auf Operationsebene
  - möglich sind aber auch
    - Funktionen
    - Codeblöcke
    - Module
    - etc.
  - Datenfluss
    - einzelne Datenwort
    - ganze Datenpakete

- Fallbeispiel Digitalfilter 1. Ordnung
  - gezeichnet als gerichteter azyklischer Graf
  - Knoten: Operationen, Dauer 1 Zeiteinheit
  - Pfeile: Datenfluss, hier ein Datenwort

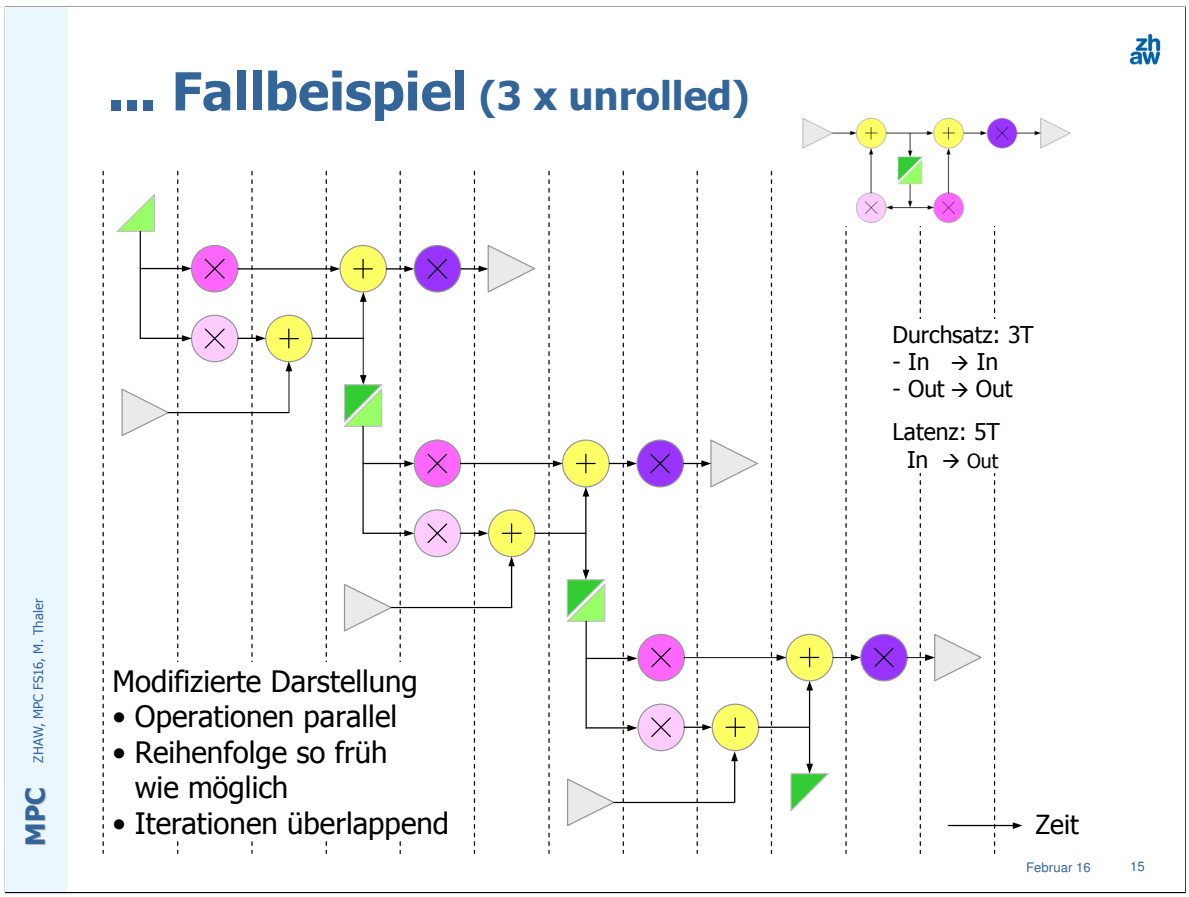
... Fallbeispiel

- **Serialisierte Darstellung**
  - pro Zeitintervall → eine Instruktion
  - gesamte Rechendauer: 9 Zeiteinheiten
  - load / store von `delay` je eine Operation



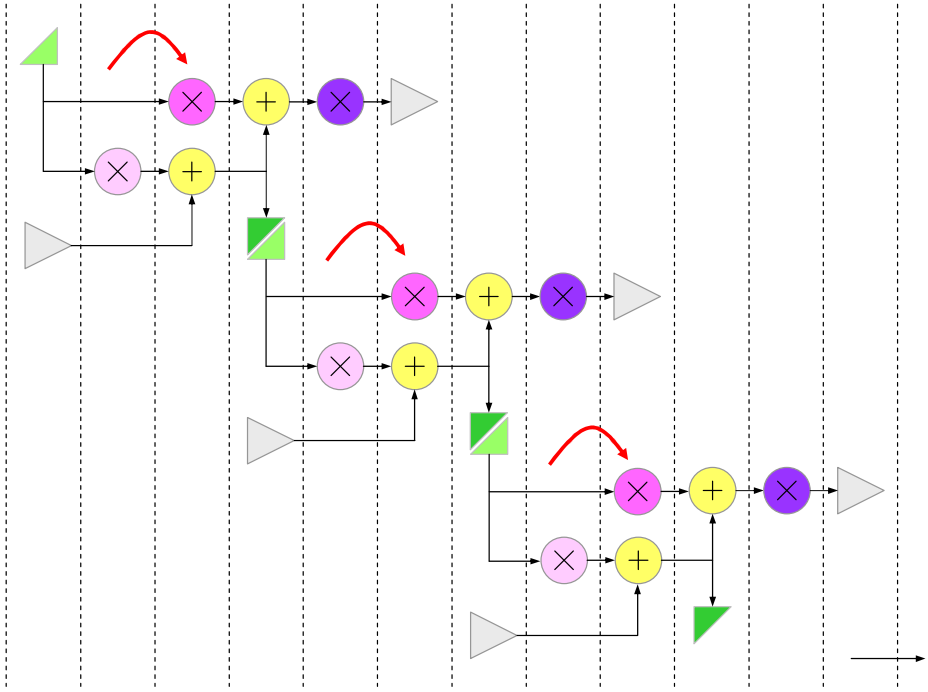


- Rekursives Digitalfilter 1. Ordnung
- Gezeichnet als gerichteter azyklischer Graf
- Pro Zeiteinheit eine Operation

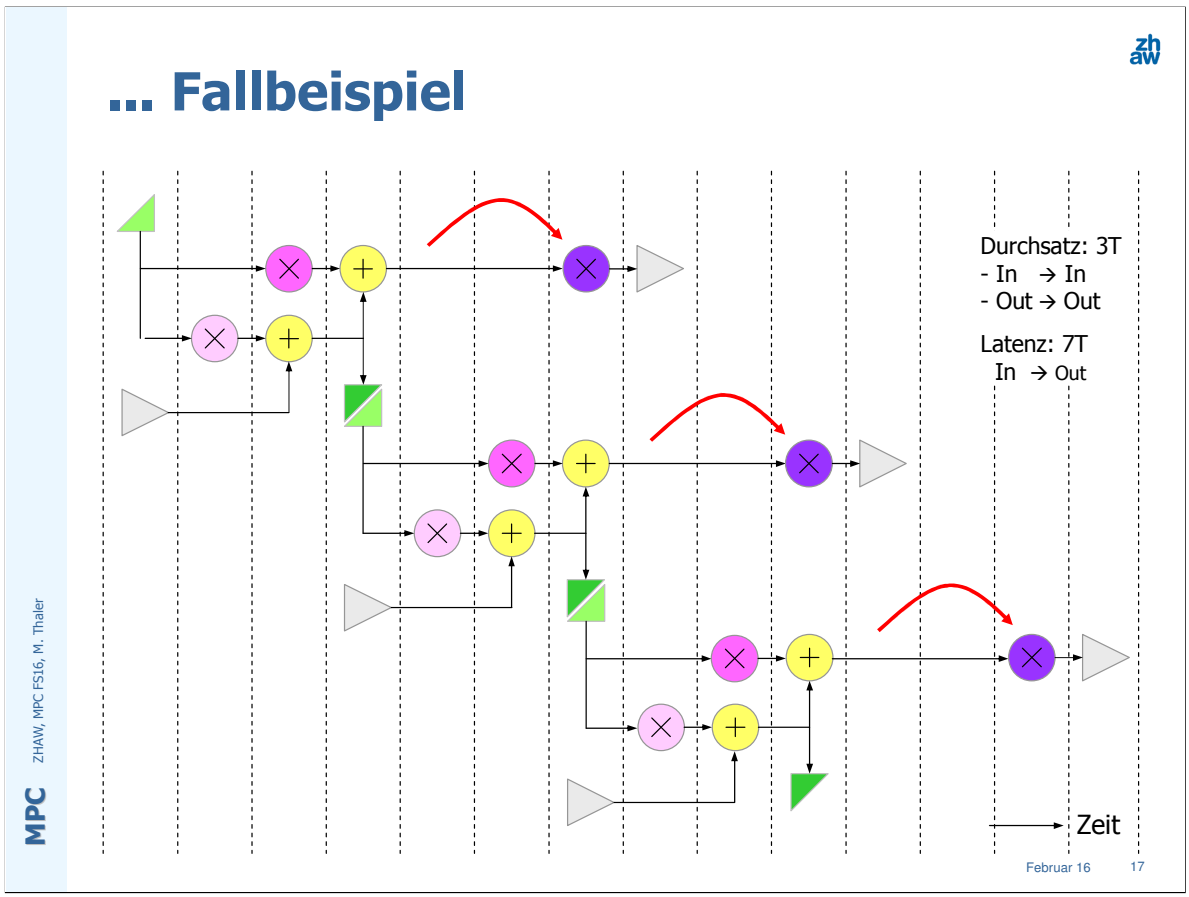


- Alle Operationen
  - frühest möglichen Zeitpunkt innerhalb einer Iteration
  - Datenflussabhängigkeiten berücksichtigt
  - Ausnahme: input (kein Einfluss auf Anordnung)
- Iterationen überlappen bei Delay
  - Annahme Variable Delay in Register
- Rechenzeit
  - Durchsatz 3 Zeiteinheiten
  - Latenz 5 Zeiteinheiten
- Benötigte Rechenressourcen
  - 2 parallele Multiplikationen
  - 1 Addition

... Fallbeispiel



- Multiplikation mit C2 um eine Zeiteinheit verschoben
- Rechenzeit
  - Durchsatz 3 Zeiteinheiten
  - Latenz 5 Zeiteinheiten
- Benötigte Rechenressourcen
  - 2 parallele Multiplikationen
  - 1 Addition



- Multiplikation mit C3 um 2 Zeiteinheiten verschoben
- Rechenzeit
  - Durchsatz 3 Zeiteinheiten
  - Latenz 7 Zeiteinheiten
- Benötigte Rechenressourcen
  - 1 Multiplikationen
  - 1 Addition
- Auslastung Rechenressourecen
  - Multiplikation: 100% 3 von 3 Zeiteinheiten
  - Addition: 66% 2 von 3 Zeiteinheiten

Take Home