Cocoma - Ein Scheduler für den Hardware/Software-Compiler Comrade

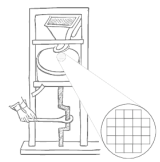
Karsten Bamberg

Diplomarbeit am Fachgebiet
Eingebettete Systeme und ihre Anwendungen (ESA)

TECHNISCHE
UNIVERSITÄT
DARMSTADT

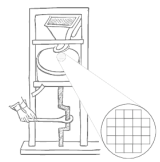
in Kooperation mit der Technischen Universität Braunschweig
Abteilung Entwurf Integrierter Schaltungen

24. April 2008



Gliederung

- Einleitung
  - Was ist ein ACS?
  - Comrade
  - Cocoma im Compilefluss von Comrade
- CMDFG
- Cocoma
- Zusammenfassung

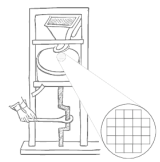


Was ist ein ACS?

- ACS = Adaptive Computing System
  - Standardprozessor (CPU)
  - Rekonfigurierbare Recheneinheit (RCU)

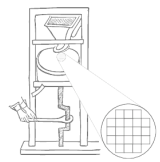
- CPU: zeitlich verteilte, serielle Abarbeitung von Operationen
- RCU: räumlich verteilte, parallele Abarbeitung von Operationen
- Taktfrequenz und Energieverbrauch bei RCU meist deutlich geringer als bei CPU

- Einsatzgebiete von ACS:
  Kryptographie, Physik, Biologie, Bild- und Videobearbeitung, ...
  $\Rightarrow$ Gebiete mit intensiven, parallelisierbaren Berechnungen

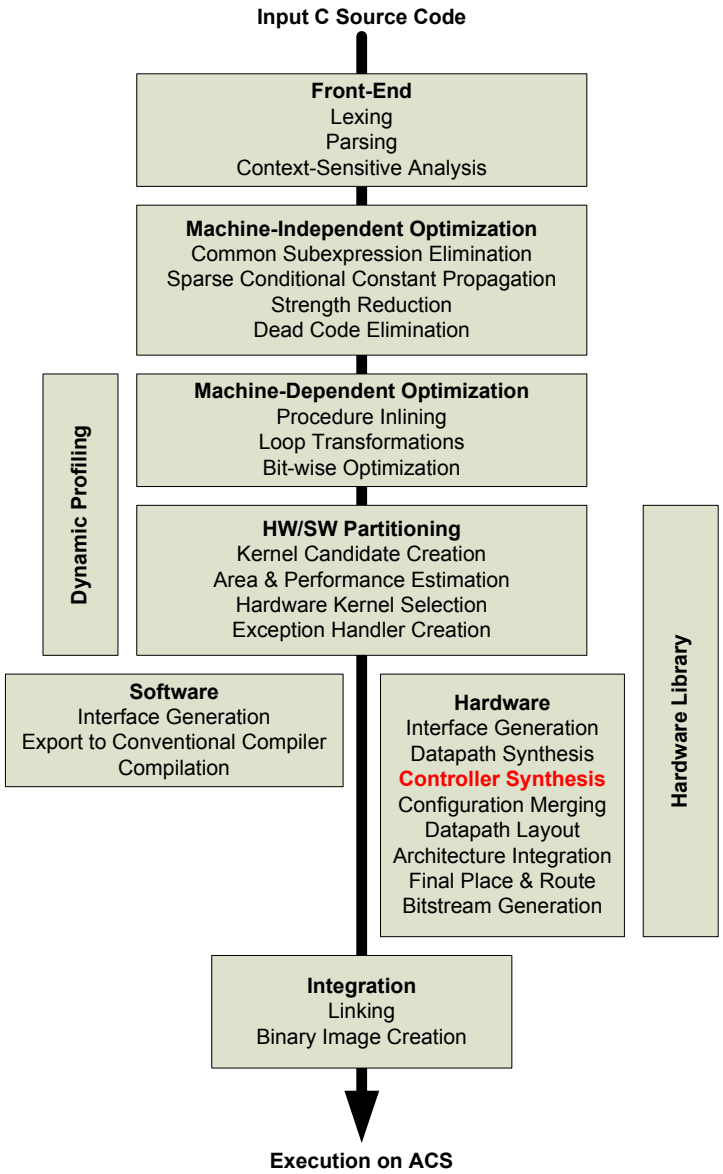


Comrade

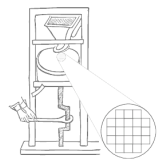
- Kompiliert C $\Rightarrow$ ACS
- Ausführung in Hardware:
  - Schleifen!
- Ausführung in Software:
  - Bibliotheksfunktionen, Floating-Point, Rekursion
- Ablauf:
  - HW/SW-Partitionierung
  - HW-Regionen
    - Zusammenhängende Regionen im CFG
    - Enthalten in HW abbildbare Statements
  - Kompilieren von SW-Teil durch gcc
  - Verbinden von SW- und HW-Teil zu ausführbarem Programm



Cocoma im Compilefluss von Comrade

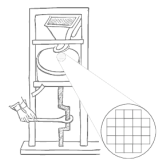


- Cocoma-Scheduling-Pass erzeugt Controller für HW-Region
- Controller steuert Ausführung der HW-Region



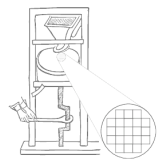
Gliederung

- Einleitung
- CMDFG
  - Spekulative Ausführung
- Cocoma
- Zusammenfassung



Control Memory Data Flow Graph (CMDFG)

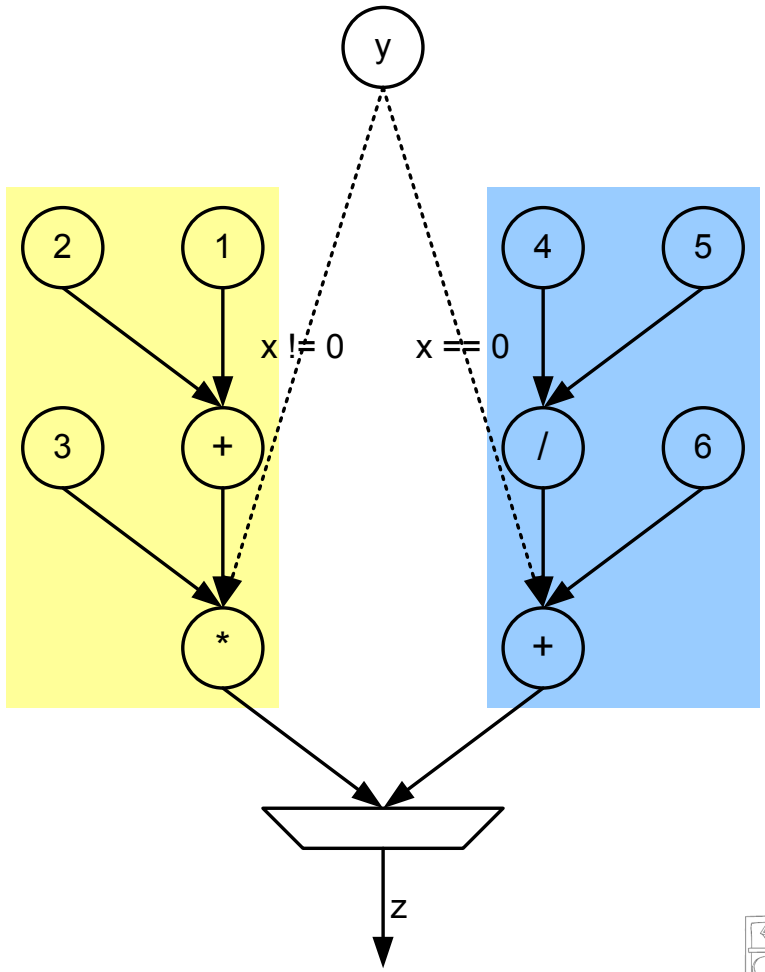
- Grundlage von Cocoma
- Wird vor Scheduling-Pass aus CFG erstellt
- Enthält Knoten und Kanten:
  - Knoten sind atomare Operationen
  - Gerichtete Kanten:
    - Daten-, Kontroll- und Speicherkanten
  - Kanten entsprechen Daten-, Kontroll- und Speicherabhängigkeiten
- Knoten starten Berechnung, sobald
  - Daten-, Kontroll- und Speicherabhängigkeiten erfüllt

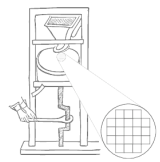


Spekulative Ausführung

- Parallele Ausführung von Then- und Else-Zweig
- Kontrollkanten auf letzten Knoten vor Mux
- Kontrollkanten entscheiden welcher Wert an Mux geht
- Speicherzugriffe aktuell nicht spekulativ

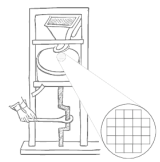
```
if (y) {
      z = 1;
      z = z + 2;
      z = z * 3;
} else {
      z = 4;
      z = z / 5;
      z = z + 6;
}
```





Gliederung

- Einleitung
- CMDFG
- Cocoma
  - Activate-Token und Cancel-Token
  - Übergangsregeln
  - Implementierung
  - Regel activate_in_next_cycle
  - Zwischenbedingung DataDependenciesFulfilled
  - Von Bedingungen zum Kernelmodul
  - Testfälle
- Zusammenfassung

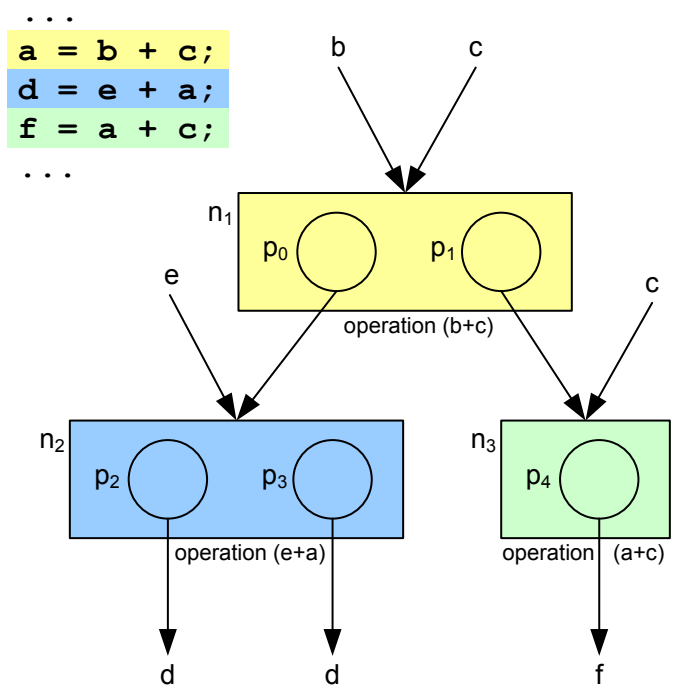


Cocoma

- Grundgerüst: CMDFG
- Jedem CMDFG-Knoten werden Plätze zugeordnet
- Anzahl der Plätze entspricht Anzahl der ausgehenden Kanten
- Jede ausgehende Kante eindeutig einem Platz zugeordnet

- Plätze enthalten Zustände:
  - 1: Activate-Token (AT)
  - 0: kein Token
  - -1: Cancel-Token (CT)

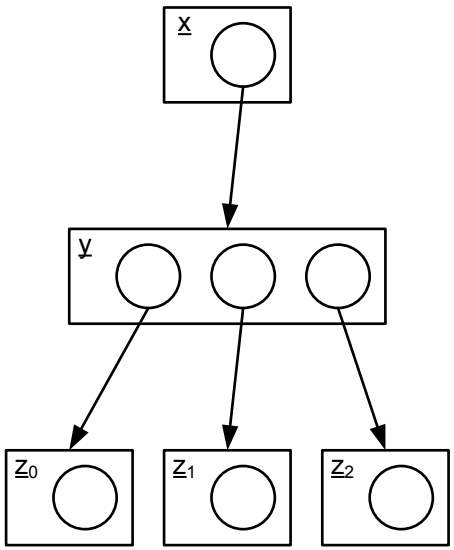
- Token bewegen sich entlang Kanten

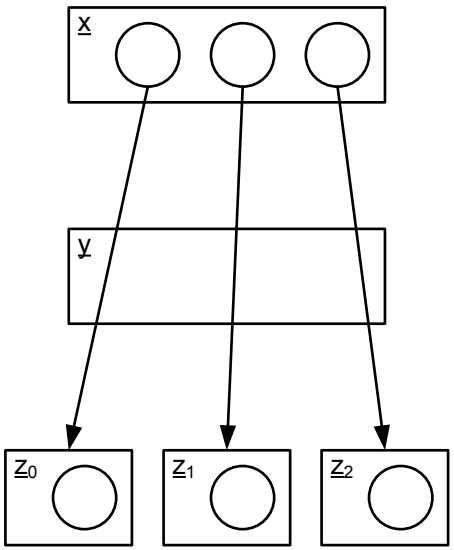


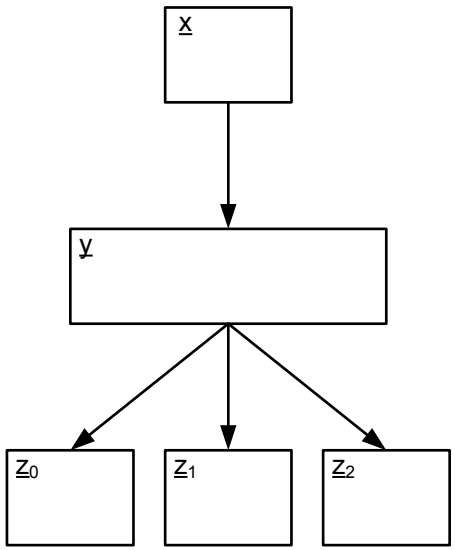
Erzeugung der Plätze

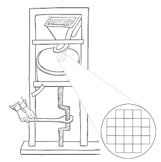
- Zuerst alle Knoten
- Danach alle Kanten

- Spezielle Knoten: DfgUnaryWiringNode, DfgBinaryWiringNode
  - Benötigen keine Register → keine Plätze
  - Rekursiver Algorithmus zum Sammeln aller Nachfolger



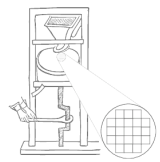




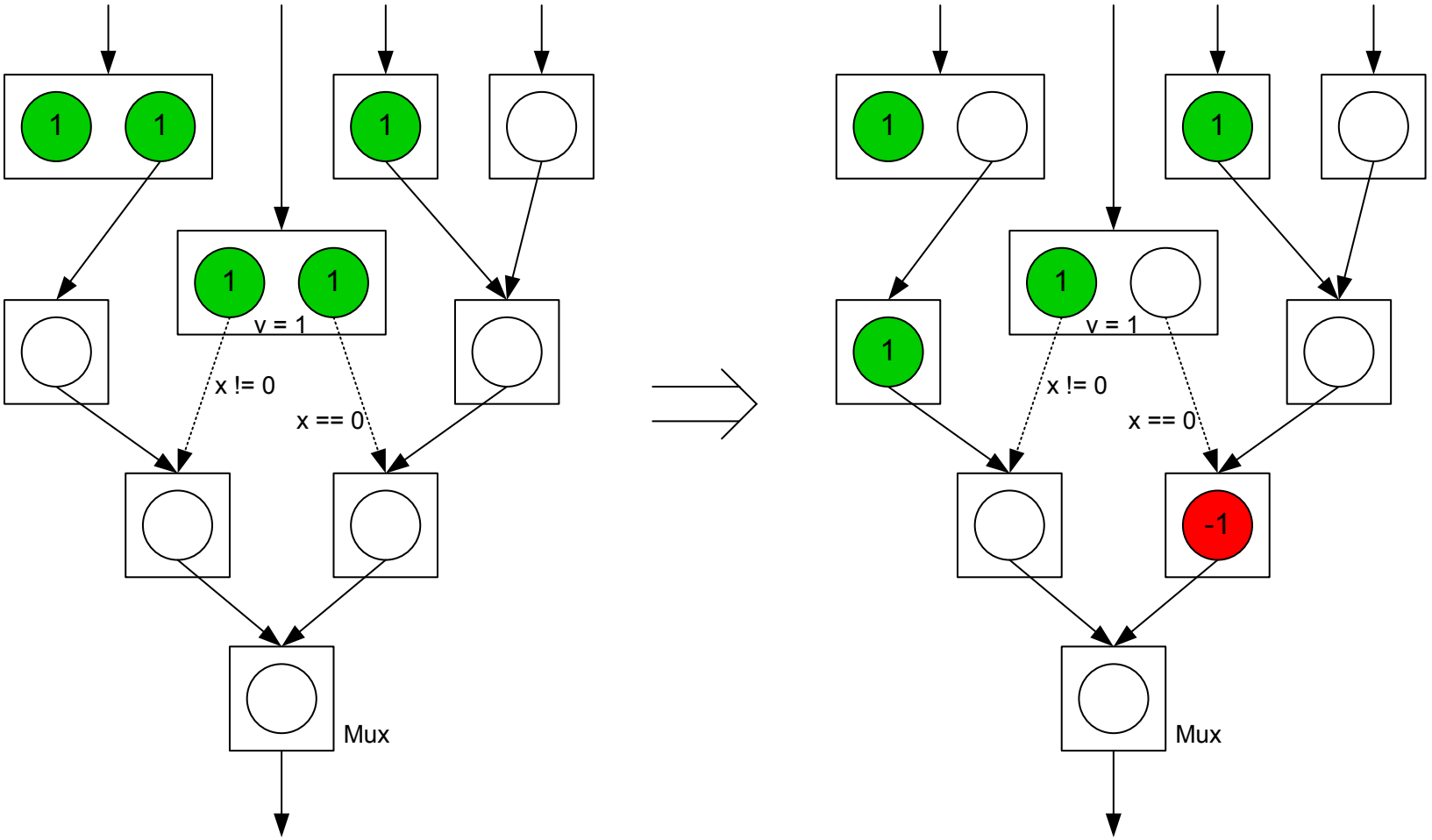


Activate-Token und Cancel-Token

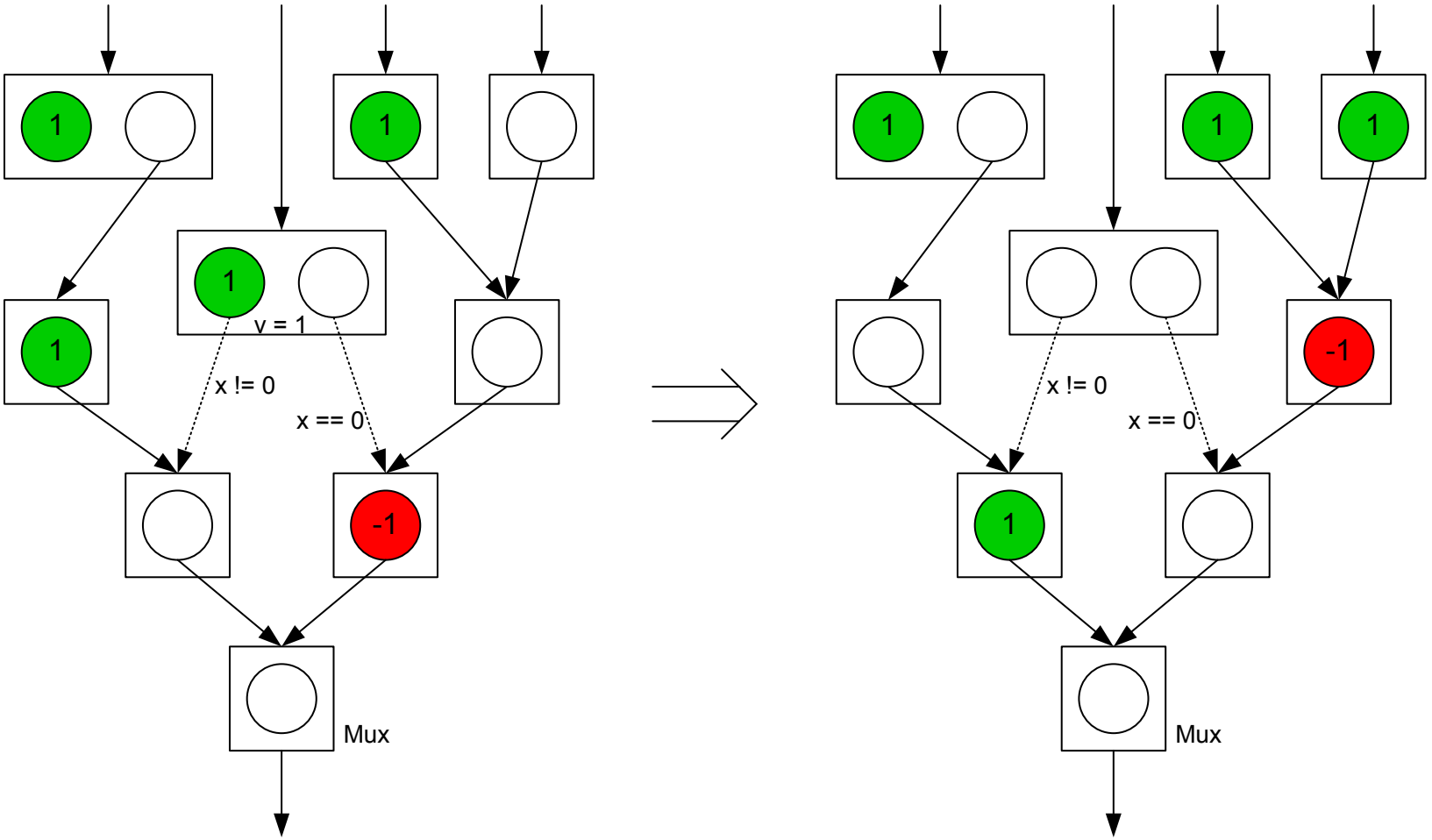
- Activate-Token
  - Funktion: Aktivierung von Operationen
  - Stehen für gültiges Datum am Datenausgang eines Knotens
  - Signalisieren bereitstehenden Eingangswert an Nachfolgerknoten
- Cancel-Token
  - Wandern entgegen dem Datenfluss
    - Rückwärts über Datenkanten
    - Löschen sich mit ATs gegenseitig aus
  - Nötig um spekulative Berechnungen abzubrechen
  - Erzeugung von CTs in Zielknoten von Kontrollkanten
    - Sobald benötigter Zweig feststeht



Activate-Token und Cancel-Token (2)



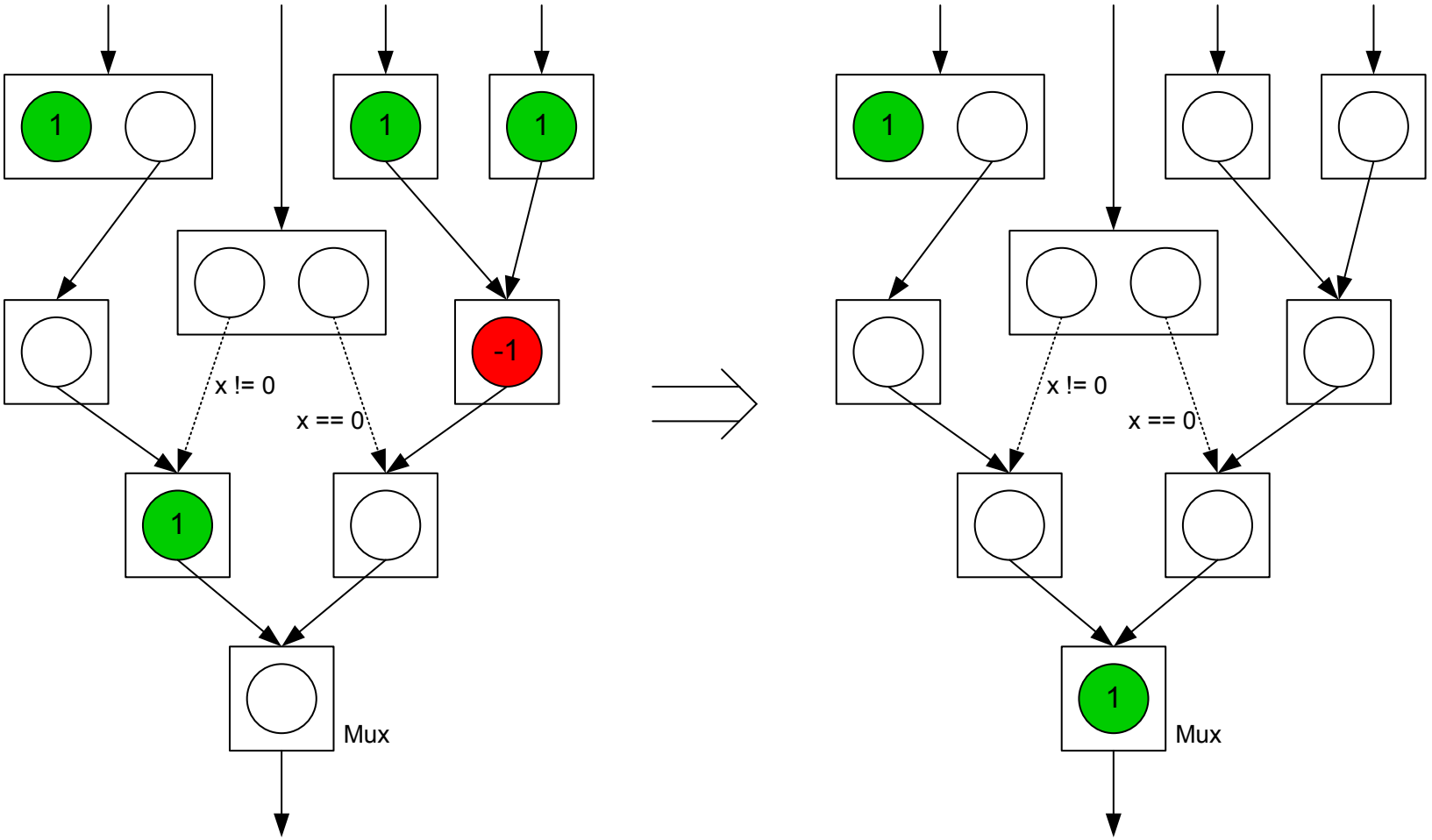
Activate-Token und Cancel-Token (3)



Takt 2

Takt 3

Activate-Token und Cancel-Token (4)

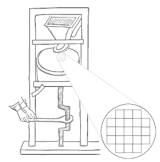


Takt 3

Takt 4

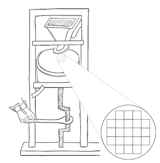
Übergangsregeln

- Bestimmen, wann Zustandsübergänge in einem Platz stattfinden:
  - 0 → 1: activate_in_next_cycle
  - 1 → 0: deactivate_in_next_cycle
  - 0 → -1: create_cancel_in_next_cycle
  - -1 → 0: remove_cancel_in_next_cycle
- Keine direkten Übergänge -1 → 1 oder 1 → -1



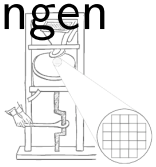
Implementierung

- Klassen
  - CocomaScheduler:
    - Steuert Ablauf, instanziert andere Klassen
  - DataKeeper:
    - Enthält Datenstrukturen mit Get- und Set-Methoden
  - FsmBuilder:
    - Erstellt Knoten, die Plätze repräsentieren
    - Eigener Graph mit Knoten und Kanten
  - MainConditionBuilder:
    - Erstellt Hauptbedingungen, z. B. $ActiveInNextCycle(fsmNode)$
  - IntermediateConditionBuilder:
    - Erstellt Zwischenbedingungen, z. B. $AllDataPredActive(dfgNode)$
  - BasicConditionBuilder:
    - Erstellt Grundbedingungen, z. B. $FsmNodeActive(fsmNode)$

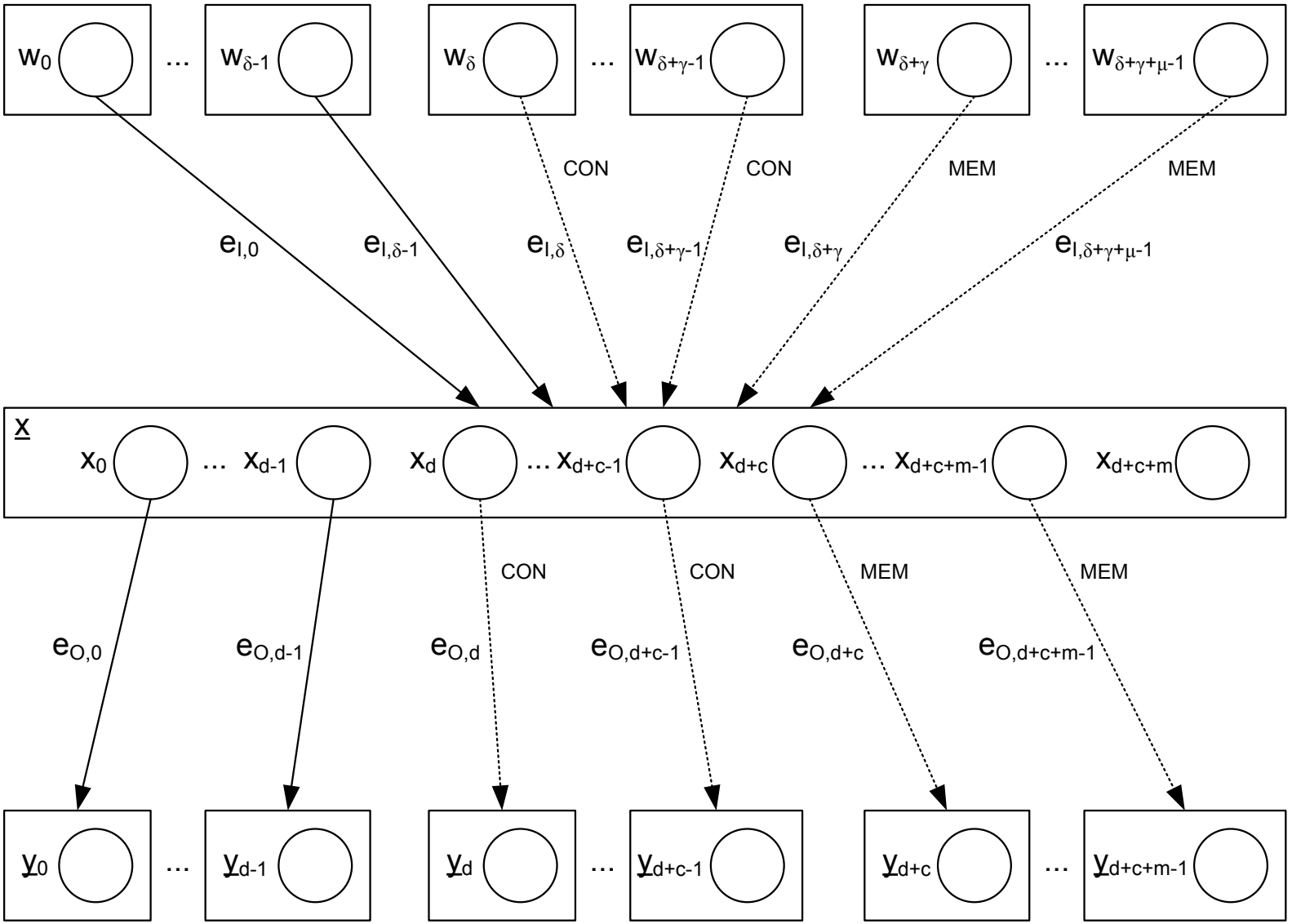


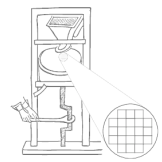
Implementierung (2)

- Aufbau von Übergangsregeln als Bedingungsbäume
  - SUIF-Expressions
- Bedingungen können logisch verknüpft werden:
  - AND, OR, NOT, EQUAL
- Hauptbedingungen:
  - Repräsentieren Übergangsregeln
  - Logische Verknüpfungen von Grund-, Zwischen- und Hauptbedingungen
- Zwischenbedingungen:
  - Sinnvoll erscheinende Zusammenfassungen von Bedingungen
  - Logische Verknüpfungen von Grund-, Zwischen- und Hauptbedingungen
  - Teilweise mehrfach benötigt



Generischer CMDFG-Knoten





Bedingung activate_in_next_cycle

- Platz ohne Token erhält im nächsten Zyklus ein AT wenn:

Informell

Kein direkter Übergang -1 $\rightarrow$ 1

und

Kein Platz im Knoten hat AT

und

Datenabhängigkeiten sind erfüllt:

(1) Falls kein Mux:

Alle Daten-Vorgänger haben AT

(2) Falls Mux:

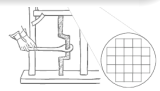
Genau ein Daten-Vorgänger hat AT

und

...

Formal

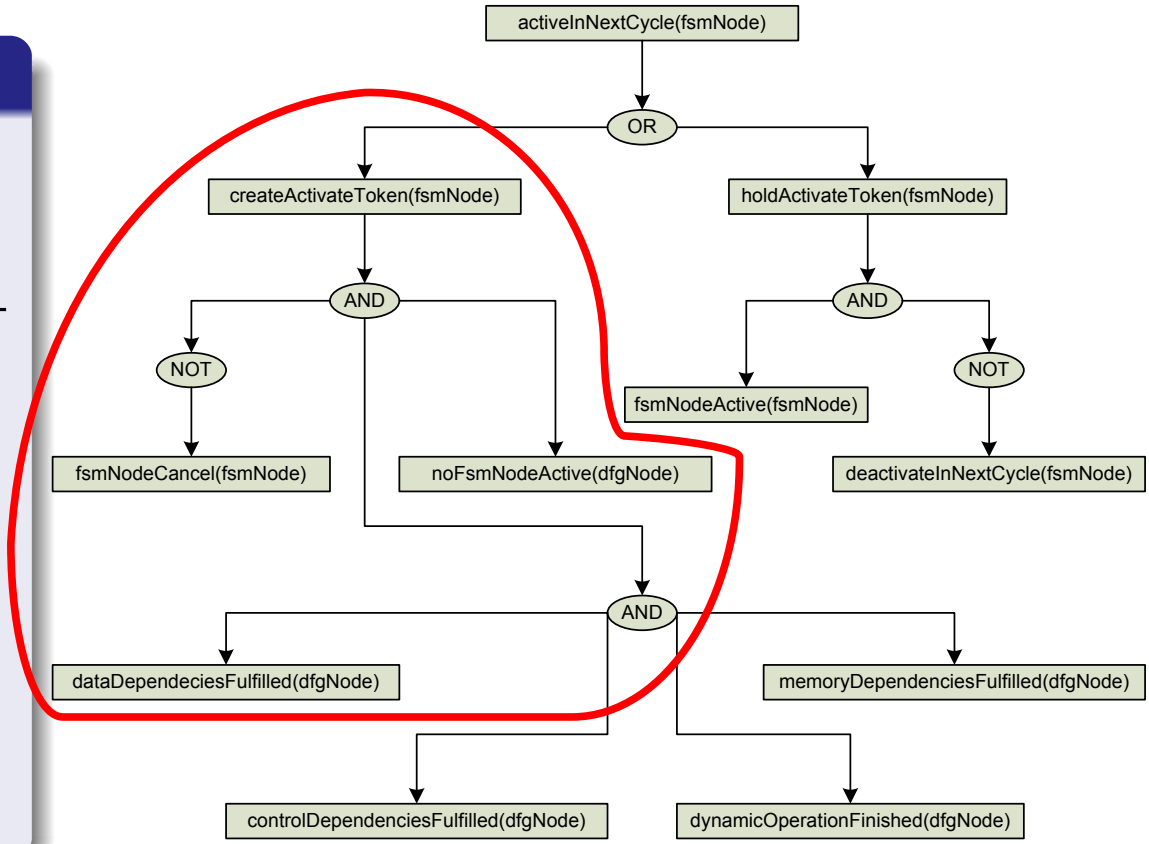
$$
\begin{gathered}
m(x_i) \neq -1 \\
\wedge \\
\forall j \in [0, d+c+m-1] : m(x_j) \neq 1 \\
\wedge \\
[(t_s(\underline{x}) \neq \text{mux} \quad \wedge \\
\forall j \in [0, \delta - 1] : m(w_j) = 1) \\
\vee \\
(t_s(\underline{x}) = \text{mux} \quad \wedge \\
\exists!\ j \in [0, \delta - 1] : m(w_j) = 1)] \\
\wedge \\
\ldots
\end{gathered}
$$

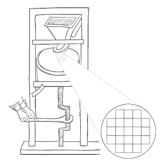


Bedingungsbaum von ActiveInNextCycle

Formal

$$
\begin{aligned}
& m(x_i) \neq -1 \\
& \wedge \\
& \forall j \in [0, d+c+m-1] : m(x_j) \neq 1 \\
& \wedge \\
& [(t_s(\underline{x}) \neq \text{mux} \quad \wedge \\
& \forall j \in [0, \delta - 1] : m(w_j) = 1) \\
& \vee \\
& (t_s(\underline{x}) = \text{mux} \quad \wedge \\
& \exists!\ j \in [0, \delta - 1] : m(w_j) = 1)] \\
& \wedge \\
& \ldots
\end{aligned}
$$





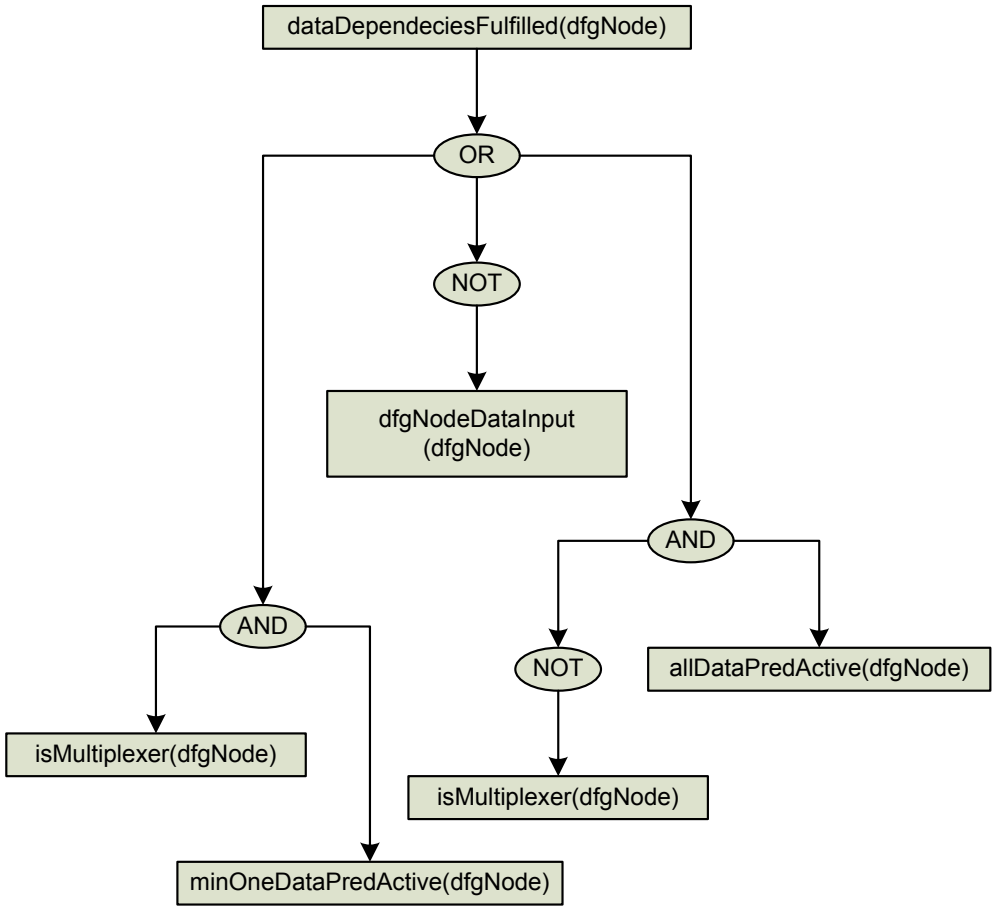
Zwischenbedingung DataDependenciesFulfilled

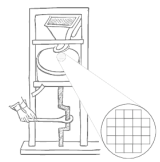
Informell

Datenabhängigkeiten sind erfüllt:
(1) Falls kein Mux:
Alle Daten-Vorgänger haben AT
(2) Falls Mux:
Genau ein Daten-Vorgänger hat AT

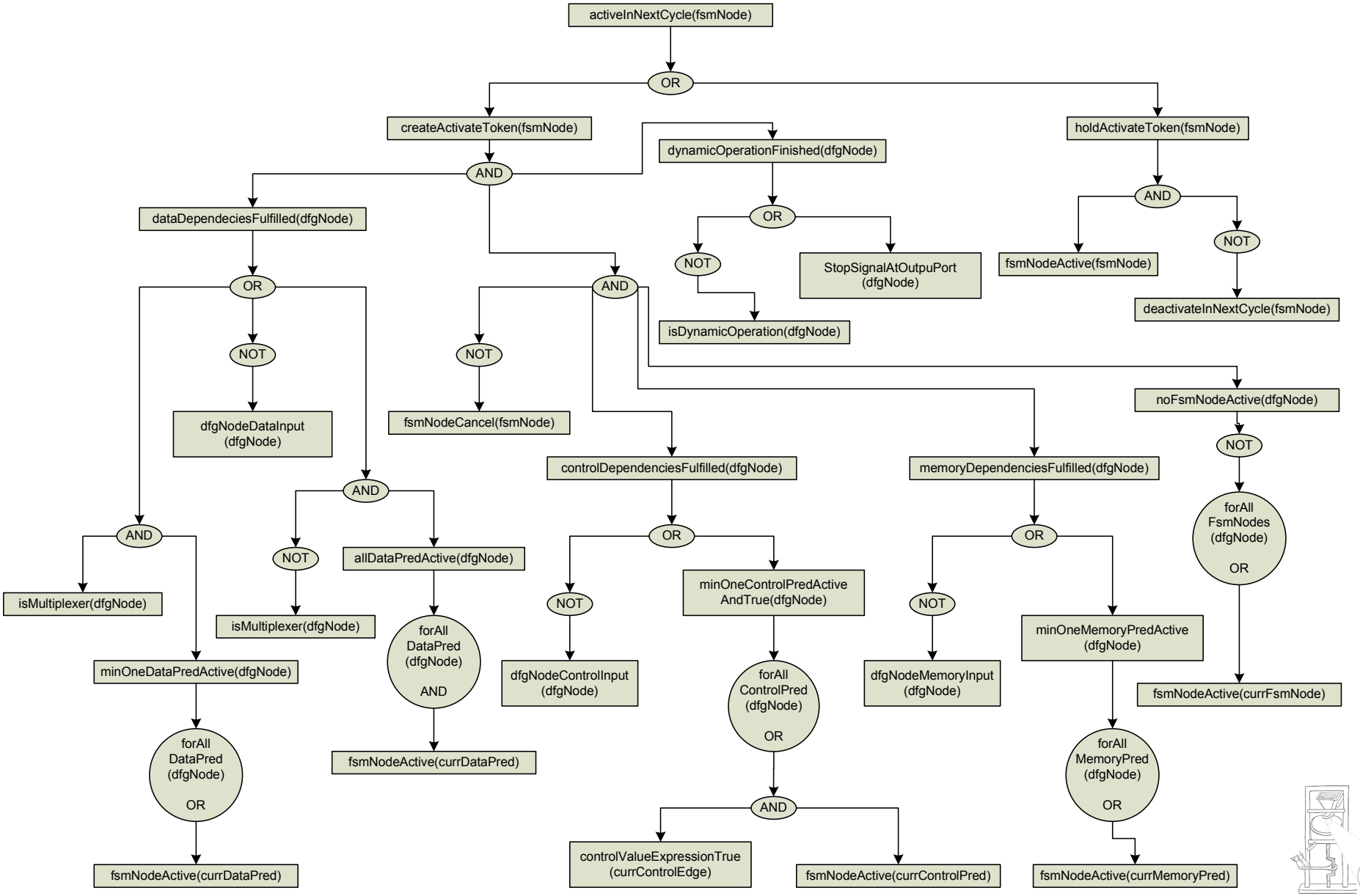
Formal

$$[(t_s(\underline{x}) \neq \text{mux} \quad \wedge$$
$$\forall j \in [0, \delta - 1] : m(w_j) = 1)$$
$$\vee$$
$$(t_s(\underline{x}) = \text{mux} \quad \wedge$$
$$\exists!\ j \in [0, \delta - 1] : m(w_j) = 1)]$$



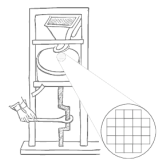


ActiveInNextCycle gesamter Bedingungsbaum



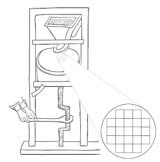
Berechnungsknoten und Multiplexer

- Statische Operationen
  - 1 Takt pro Berechnung
  - Clock-Enable-Eingang → Ergebnis zur nächsten Taktflanke
  - Bedingung ComputeStaticOperation
- Dynamische Operationen
  - Mehrere Takte pro Berechnung → Variable Laufzeit
  - Start-Eingang → Startet Berechnung
  - Done-Ausgang → Signalisiert Berechnungsende
  - Bedingung StartDynamicOperation
- Multiplexer
  - Leitet jedes gültige Datum weiter
  - Select-Signal für Vorgänger aktiv, wenn Vorgänger AT hat
  - One-Hot-Select



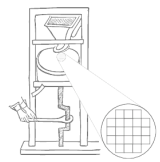
Von Bedingungen zur Hardware

- Comrade erzeugt Verilog-Datei für HW-Teil
- Register für Tokenplätze
- Methode insert_condition()
  - Einfügen von Grund-, Zwischen- und Hauptbedingungen
    - Als wires in Verilog-Datei
  - Einfügen von assign-Befehlen für Bedingungen in Verilog-Datei
    - Logische Verknüpfungen von Bedingungen



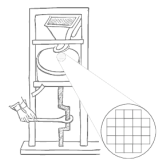
Testfälle

- Test von HW-Teilen in Simulator
- Vergleich der Laufzeiten mit Pentium 4 System
  - Pentium 4: 3,0 GHz
  - angenommene Taktrate für HW-Lösung: 100 MHz
- Kleine Testfälle mit wenig Parallelität
  - Pentium 4 oft deutlich schneller
- Parallel-Testfall
  - 50 unabhängige Additionen $\rightarrow$ parallel ausführbar
  - HW-Lösung etwa 4% schneller



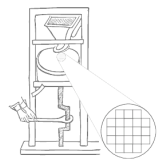
Gliederung

- Einleitung
- CMDFG
- Cocoma
- Zusammenfassung



Zusammenfassung

- Cocoma-Scheduler erzeugt Controller für HW-Teil in Comrade
- Cocoma
  - Basiert auf CMDFG
  - Verwendet Activate-Token und Cancel-Token
  - Übergangsregeln als Bedingungsbäume implementiert
  - Haupt-, Zwischen- und Grundbedingungen
    - Modularität
  - Mathematische Grundlage möglichst genau umgesetzt
  - Ermöglicht gute Wartbarkeit
  - Ersetzt Implementierung des StraightSchedulers



Ende

Zeit für Fragen