Christian-Albrechts-Universität zu Kiel

Diplomarbeit

Beschreibung des Kiel Esterel Prozessors in Esterel

kep$^{\varepsilon}$

Malte Tiedje

29. Januar 2008

Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

betreut durch:
Claus Traulsen

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

Moderne Mikroprozessoren, die vor allem in PCs zu finden sind, werden für universelle Anwendungsbereiche entwickelt und können eine große Menge von Befehlen pro Sekunde verarbeiten. Diese Gruppe von Prozessoren sind teuer und verbrauchen sehr viel Energie (5 mal mehr pro cm^2 als eine Kochplatte). Schon aus diesen Gründen können diese Prozessoren nicht in Geräten eingesetzt werden, die autonom ohne ständige Stromversorgung auskommen müssen und bei denen der Preis des Gerätes nicht vom Prozessor bestimmt werden soll. Häufig sind diese Systeme *eingebettete Systeme*, für die leichtgewichtigere Prozessoren oder Mikrocontroller existieren. Oftmals werden überarbeitete Designs früherer Prozessorgenerationen verwendet, die ihrerzeit für universelle Anwendungen entwickelt worden sind, aber heute vergleichsweise preiswert und sparsam sind. Programme für diese Plattformen werden üblicherweise hardwarenah in *C* oder Assembler geschrieben. Durch handgeschriebenen Code können sich Programmierfehler einschleichen, die in sicherheitskritischen Anwendungen zu katastrophalen Folgen führen können. Zudem wird bei es zeitkritischen Anwendungen wie z. B. einer Airbagsteuerung zunehmend schwieriger ein exaktes Zeitverhalten der ausgeführten Programme zu garantieren. Derartige Systeme gehören meistens auch zu den *reaktiven Systemen* [5]. Diese werden durch ihre Eingaben dominiert und müssen in vorhersagbarer Zeit Ausgaben liefern. Mit synchronen Sprachen [3] lassen sie sich präzise beschreiben. Diesen Sprachen liegt ein diskretes Zeitmodell zu Grunde, welches vom realen Zeitmodell abstrahiert. In der Regel wird aus diesen Sprachen *C*-Code erzeugt, der sicherer ist als handgeschriebener. Jedoch ist der Code gerade bei stark nebenläufigen Problemen nicht so effizient und daher häufig zu langsam oder zu umfangreich.

Ein weiterer Ansatz ist es, nicht einen bestehenden Prozessor für diese Aufgaben zu wählen, sondern für genau diesen Anwendungsfall eigene Hardware zu entwickeln. Diese wird als *Application-Specific Integrated Circuit* (*ASIC*) Design bezeichnet. Der Vorteil besteht darin, dass dieser Schaltkreis ein präzises Zeitverhalten hat und meist konkurrenzlos schnell ist. Jedoch benötigt man sehr hohe Stückzahlen, um auf ein Preisniveau zu kommen, das sich mit preiswerten Mikrocontrollern messen kann. Desweiteren ist dieser Schaltkreis nur genau für eine Aufgabe entwickelt worden und kann kaum auf neue Anforderungen oder Veränderungen der Umgebung angepasst werden.

Es wird also ein Prozessor benötigt, der sowohl ein exaktes Zeitverhalten hat als auch flexibel anpassbar ist. Dieser Prozessoren kann in die Klasse der *Application-Specific-Instruction-Processor* (*ASIP*) eingeordnet werden und beschreibt einen Prozessor, der nicht mehr universell einsetzbar ist, sondern nur für bestimme Anwendungsgebiete eigene Befehle bereitstellt. In diese Gruppe ist auch der *Kiel Esterel*

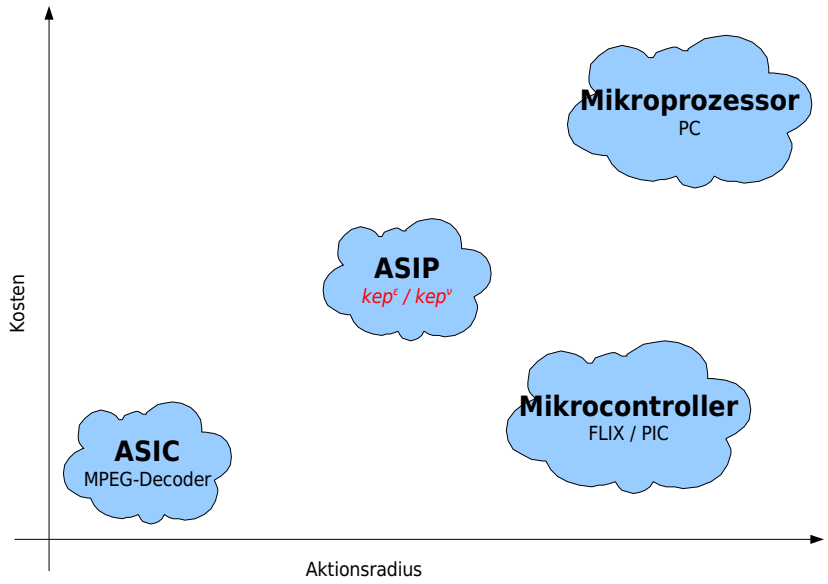


Abbildung 1.1: Übersicht verschiedener Steuerungssysteme für eingebettete Systeme

Prozessor (*kep*) [26] einzuordnen. In Abb. 1.1 wird eine Übersicht der verschiedenen Prozessoren oder Schaltkreise gegeben.

Der Befehlssatz des *kep* entspricht semantisch der synchronen Sprache Esterel [6] und vereint damit zwei wichtige Eigenschaften der anderen Ansätze: deterministisches Zeitverhalten und Flexibilität.

Ziel dieser Arbeit ist die Beschreibung des Kiel Esterel Prozessors in Esterel (kep^{ε}). Dabei soll untersucht werden, wie sich die synchrone Sprache Esterel zum Entwurf komplexer Hardware eignet. Bei der klassische Hardwareentwicklung wird meist mit Hardwarebeschreibungssprachen (HDL) wie *VHDL* und *Verilog* entwickelt, die jedoch nicht über eine strikte synchrone Semantik verfügen.

Eine verbreitete These aus dem Umgang mit Programmiersprachen besagt, dass man eine Sprache erst wirklich versteht, wenn man für diese Sprache einen Compiler in dieser Sprache geschrieben hat. Analog lässt sich sagen: Man hat Esterel erst wirklich verstanden, wenn man einen Prozessor für Esterel in Esterel entwickelt hat.

In dem folgenden Kapitel werden die Grundlagen für die Entwicklung mit Esterel beschrieben und ein Einblick in die verwendeten Werkzeuge gegeben. In Kapitel 3 wird eingehend die Implementierung beschrieben, und in Kapitel 4 werden die Ergebnisse des erstellten Prozessors quantifiziert und gegen die VHDL Implementierung verglichen.

1.1 Andere Arbeiten

Das Forschungsgebiet über reaktive Prozessoren ist relativ jung und begann 2002 mit dem ersten Esterel Prozessor ReFLIX [34] von Salcic et al.. Dieser Ansatz beschreibt eine Erweiterung des Opensource Prozessors FLIX um einen reaktiven Befehlssatz, der Esterel ähnliche Instruktionen beschreibt. 2004 wurde durch Chow et al. der RePIC [12] vorgestellt, bei dem der FLIX Kern durch einen PIC [31] Kern ersetzt wurde. Diese beiden Ansätze werden von Küçükçakar [25] als *Patched Processor* klassifiziert und bieten alleine nur eingeschränkte Unterstützung für Nebenläufigkeit. Daher wurde für den RePIC eine Multiprozessor-Architektur entworfen, die volle Unterstützung für Nebenläufigkeit bietet. Diese ist im EMPEROR [14] umgesetzt worden. 2004 wurde auch mit der Entwicklung des kep^{v} [27] in *VHDL* durch Xin Li begonnen. Bereits in Version 2 (2005) [29] bot der kep^{v} volle Unterstützung für sequenzielle Esterel Programme und in Version 3 (2006) [28] wurde der Entwurf mit Hilfe von Threads um Nebenläufigkeit erweitert. In der Dissertation von Li [26] wurde die Version 4 abschließend vorgestellt, die auch in dieser Arbeit als Vorlage und zu Vergleichszwecken dient. In der Arbeit von Gädtke [18] wurde zudem der kep^{v} durch programmspezifische Hardware erweitert, mit der die Verarbeitungsgeschwindigkeit verbessert werden konnte.

Der kep^{ε} ist nicht der erste Prozessor der ausschließlich in Esterel entwickelt wurde. Esterel-Technologies [35] wirbt mit mehreren erfolgreichen Projekten, jedoch wird kein tiefer Einblick gewährt. Ein besseres Beispiel für einen in Esterel geschriebenen Prozessor ist die Arbeit von Weber [36]. Im Rahmen einer Diplomarbeit wurde hierbei ein RISC Prozessor entwickelt und damit die Leistungsfähigkeit von Esterel als Hardwarebeschreibungssprache eingehend untersucht. Durch die synchrone Semantik konnte insbesondere die formale Verifikation in dem Projekt erfolgreich eingesetzt werden.

2 Grundlagen

Reaktive Systeme stehen in ständiger Interaktion mit ihrer Umgebung, wobei die Umgebung das System bestimmt. Von reaktiven Systemen wird verlangt, das sie zu gegeben Eingaben immer die gleichen Ausgaben liefern und das immer in der gleichen Zeit. Solche Systeme lassen sich gut mit *synchronen Sprachen* [3] beschreiben, da sie deterministisch und ein diskretes Zeitverhalten haben. Die bekanntesten synchronen Sprachen sind Signal [21], Lustre [23] und Esterel [6]. Im Folgenden wird eine kurze Einführung in Esterel gegeben und besonders auf die Hardwareerzeugung eingegangen. Darüberhinaus wird Esterel Studio vorgestellt und Probleme bei der Entwicklung mit Esterel betrachtet.

2.1 Einführung Esterel

Die imperative Programmiersprache Esterel ist eine synchrone, nebenläufige Sprache, die vornehmlich für die Steuerung von reaktiven Systemen entwickelt wurde. Esterel liegt die *synchrone Hypothese* [7] zu Grunde. Dieses Modell zerlegt Berechnungen und Verhalten in diskrete Ausführungsschritte, sog. Instanzen. Somit wird die Menge der Anweisungen in Esterel in zwei Gruppen unterschieden: instantan oder verzögert (*delayed*). Instantane Anweisungen verbrauchen keine Zeit, während verzögerte Anweisungen die Ausführung für die aktuelle Instanz beenden. Die meisten Befehle wie loop, emit oder present gehören zu den instantanen Anweisungen, während pause, await und every zu den verzögert Anweisungen gehören.

Signale

Für die Kommunikation mit der Umgebung verfügt Esterel über Ein- und Ausgabesignale. Diese Signale bilden die Schnittstelle eines Esterel Programms. Der Status eines Signals ist für jede Instanz entweder gesetzt (*present*) oder nicht gesetzt (*absent*). Dies wird als *logische Kohärenz* bezeichnet. Neben den ein- und ausgehenden Signalen gibt es *lokale* Signale, die für die Kommunikation zwischen verschiedenen Programmteilen verantwortlich sind. Zusätzlich zu dem aktuellen Status eines Signals lässt sich über pre auf den Status des Signals in der vorherigen Instanz zugreifen. Signale können entweder ohne Wert (*pure*) oder mit Wert (*valued*) behaftet sein. Im letzteren Fall sind in Esterel v5 verschiedene Datentypen wie `integer, float` oder `double` definiert.

Der Status von Signalen kann von anderen Signalen abhängen. Dies ist unkritisch, solange der Abhängigkeitsgraph nicht zyklisch ist. Die meisten Compiler für Esterel,

darunter auch `strl2kasm` [10], schreiben vor, dass die Programme azyklisch sein müssen.

Unterbrechnungen

Eine weitere wichtige Gruppe von instantanen Befehlen sind die Anweisungen zum Anhalten oder Unterbrechen des Kontrollflusses, die sog. *preemption* Instruktionen. Ein strong abort bricht die Ausführung seines Geltungsbereiches in der gleichen Instanz ab, wenn das auslösende Signals anliegt. Ein weak abort hingegen beendet noch die laufende Instanz. Wenn der Kontrollfluss nur vorübergehend angehalten aber nicht abgebrochen werden soll, wenn ein Signal anliegt, kann dies mit suspend ausgedrückt werden.

Delays

Da reaktive Systeme von ihren Eingaben abhängen, enthält Esterel spezielle Befehle, mit denen auf das Auftreten von Signalen gewartet werden kann. Bei diesen sog. Delayausdrücken werden drei Klassen unterschieden:

standard delays warten auf das Auftreten eines Signals und werden nicht in der initialen Instanz geprüft.

immediate delays , wie *standard delays*, können jedoch bereits in der initialen Instanz terminieren.

count delays entsprechen den *standard delays*, jedoch wird auf das mehrfache Auftreten eines Signals gewartet.

Mit dem ||-Operator wird Nebenläufigkeit in Esterel beschrieben. Die Ausführung ist beendet, wenn alle gleichzeitig ausgeführten Blöcke terminiert sind. Signale können von nebenläufig ausgeführten Blöcken sowohl geschrieben als auch gelesen werden, wobei Variablen starken Restriktionen unterliegen, um Schreib-Lesekonflikte zu umgehen. Variablen dürfen nur innerhalb eines nebenläufigen Blocks gelesen oder geschrieben werden. Erstreckt sich der Definitionsbereich über mehrere nebenläufige Böcke, so kann eine Variable nur gelesen werden.

Esterel enthält auch Befehle zur Behandlung von Ausnahmebedingungen. Über trap wird eine Ausnahmebedingunge deklariert, und über exit wird der Geltungsbereich instantan abgebrochen. Sind Traps parallel definiert und treten die Bedingungen gleichzeitig auf, so erhält die äußerste Trap Priorität.

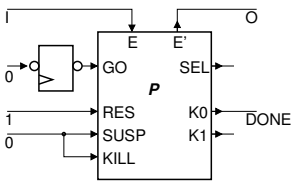
In Tabelle 2.1 sind die wesentlichen Esterel Instruktionen, die sog. *Kernel-Befehle*, angegeben. Weitere Befehle lassen sich aus diesen komponieren: In Abb. 2.2 ist abort definiert über trap und suspend. Sobald das Signal `S` auftritt, wird der Bereich p angehalten und gleichzeitig die Trap ausgelöst. Wenn p normal terminiert, wird auch die Trap ausgelöst, um die überwachende Schleife zu verlassen.

Esterel Quelle	**Beschreibung**
;	Sequenzen Operator
||	Parallel Operator
pause	Der Kontrollfluss wird angehalten und startet wieder in der nächsten Instanz.
emit S	Signal `S` wird ausgegeben und ist in der aktuellen Instanz gesetzt.
present *S* **then** *p* **else** *q* **end present**	Wenn das Signal `S` gesetzt ist, wird p ausgeführt, ansonsten q.
loop p **end loop**	Führt p unendlich oft nacheinander aus bis von außen abgebrochen wird. Dafür darf der Inhalt von p nicht instantan sein.
suspend *p* **when** S	Wenn Signal `S` auftritt, wird p in der Ausführung angehalten.
nothing	keine Aktion
signal S **in** p **end signal**	Definiert ein neues lokales Signal `S` mit Geltungsbereich p. Bisher definierte (globale) Signale `S` werden überschrieben, auch wenn diese gerade gesetzt sind. `S` ist nach der Definition nicht gesetzt, bis es durch `emit S` gesetzt wird.
trap *T* **in** ... **exit** *T* ... **end trap**	Der `exit T` Befehl bricht die Ausführung der **trap** ab, wenn es aufgerufen wird.

Tabelle 2.1: Esterel Kernel Befehle

2.1.1 Schaltkreissemantik

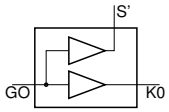
Für Esterel sind mehrere Semantiken formalisiert [5]. Für die Hardwareentwicklung ist die Schaltkreissemantik am wichtigsten, mit der Esterel Programme in Schaltkreise von Logikgattern übersetzt werden können. Dies kann sehr effizient geschehen, indem nebenläufige Abschnitte in parallele Schaltkreise übersetzt werden. Diese Übersetzung ist von Berry [5] entworfen worden und gibt für jeden Esterelbefehl eine Übersetzungsregel an. Einige dieser Regeln und deren Repräsentation (Abb. aus [30]) werden im Folgenden vorgestellt, um einen Eindruck zu vermitteln, wie mit Esterel Hardware erzeugt werden kann.



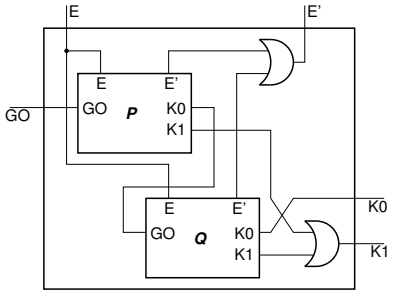

```
module MAIN:

input I;
output O;

P

end
```

Jedes Modul P wird in einen Block übersetzt, der aus der Schnittstellenbeschreibung mit Eingaben E und Ausgaben E' besteht. Darüberhinaus gibt es Eingänge RES, SUSP, KILL und SEL für Unterbrechungen und Abbrüche des Programms und den Ausgänge K0, der anzeigt, ob das Programm noch läuft oder normal terminiert.

K1 steht für eine Menge K_n von Ausgängen, wobei n für die Anzahl der gleichzeitigen Traps steht. Für jede ausgelöste Trap wird der dazugehörige Ausgang gesetzt, um anzuzeigen, dass das Programm abgebrochen worden ist. Gestartet wird das Programm über den GO Eingang, der zu Beginn über das sog. *boot register* gesetzt wird. Der Taktgeber ist mit jedem Register verbunden und betreibt damit den gesamten Schaltkreis synchron. In der Abbildung ist der Eingang des Taktgebers nicht explizit dargestellt. Da jedes Programm mit den genannten Pins ausgestattet ist, werden diese im Folgenden zur besseren Lesbarkeit weggelassen.

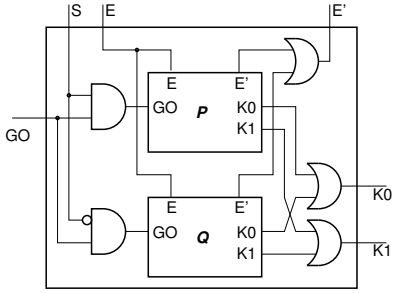


emit S



P ; Q

Das Emittieren eines Signals ist instantan und setzt nur den Ausgang S', der dann als Eingang weiter verwendet werden kann. Sollen zwei Programme P und Q hintereinander ausgeführt werden, so wird einfach mit Hilfe des Ausgangs $K0$ von Programm P der GO von Programm Q gesteuert. Für die äußere Sicht werden beide Ausgänge $K1$ durch ODER verknüpft. Gleiches gilt auch für die Ausgänge E'.



present S **then**
  P
else
  Q
end

Für die konditionale Ausführung von Programmen ergibt sich ein ähnliches Bild: Der Eingang E wird zu beiden Teilen propagiert, während der Zustand des Signals S für die UND Verknüpfung mit GO negiert wird. Damit wird entweder P oder Q ausgeführt.

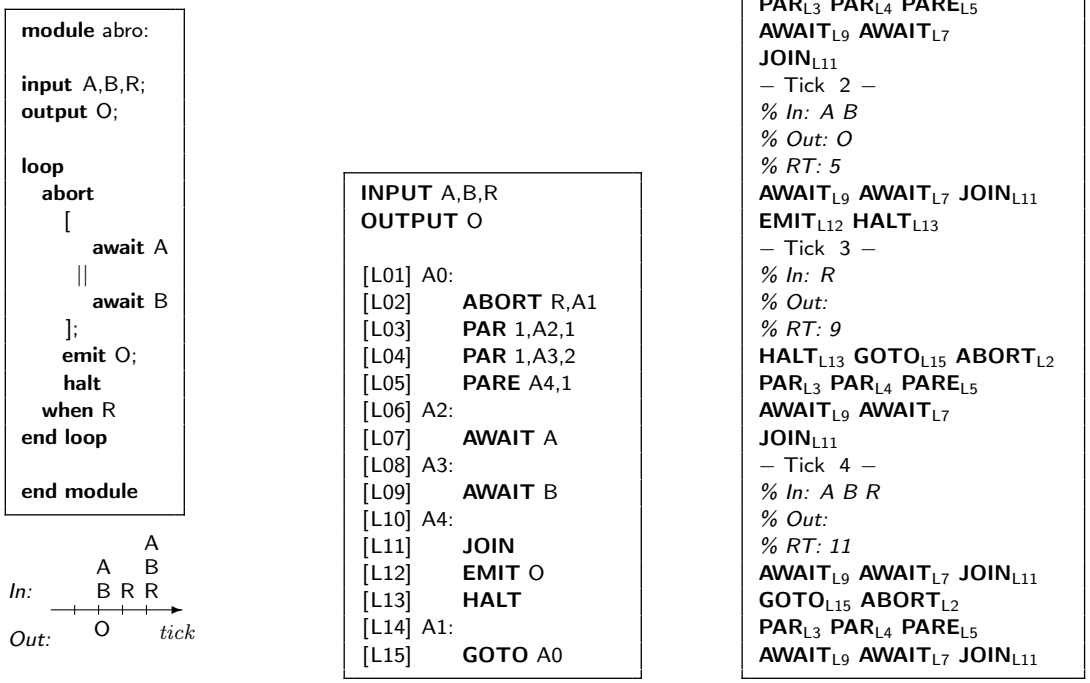
Für die gleichzeitige Ausführung werden zwei Programme P und Q parallel ausgeführt und jeweils die Ausgänge K mit einem sog. *Synchronizer* verbunden. Vereinfacht gesprochen wartet der Synchronizer, bis beide Programme terminiert sind, um dann selber die Terminierung anzuzeigen.

Esterel Quelle	***kep* Assembler**	**Bemerkungen**
emit *S* [(*val*)]	EMIT *S* [, {#*data*|*reg*}]	Emittiere Signal *S (Optional mit Wert)* .
present *S* **then** p **else** q **end present**	PRESENT *S*, *elseAddr* P GOTO *endAddr* *elseAddr*: Q *endAddr*:	Springe zur *elseAddr*, wenn *S* nicht gesetzt ist. GOTO wird nicht angegeben, wenn es keinen *else* Fall gibt.
[weak] **abort** p **when** [immediate, *n*] *S*	[LOAD _COUNT,*n*] [W]ABORT[I] *S*, *endAddr* P *endAddr*:	Um Unterbrechungen für n Ticks zu verzögern, wird die interne Variable _COUNT gesetzt.
suspend ... **when** [immediate, *n*] *S*	[LOAD _COUNT,*n*] SUSPEND[I] *S*, *endAddr* ... *endAddr*:	
trap *T* **in** ... **exit** *T* ... **end trap**	*startAddr*: ... EXIT *exitAddr startAddr* ... *exitAddr:*	Beendigung durch eine Trap, *startAddr*/*exitAddr* geben den Geltungsbereich an. Im Unterschied zu GOTO, prüfe gleichzeitige EXITs und beende eingeschlossene ||.
pause	PAUSE	Warte n mal auf das Signal S. AWAIT TICK ist äquivalent zu PAUSE.
await [immediate, *n*] *S*	[LOAD _COUNT, *n*] AWAIT [I] *S*	
sustain *S* [(*val*)]	SUSTAIN *S* [, (#*val*|*reg*)]	Gebe Signal *S(opt. m. Wert)* dauerhaft aus.
halt	HALT	Hält das Programm an.
nothing	NOTHING	Keine Operation. Manchmal zur Unterscheidung von Sprungmarken nötig.
loop ... **end loop**	*addr*: ... GOTO *addr*	Springe zu *addr*. Der Inhalt der Schleife darf nicht instantan sein.
[p_1 || ⋮ || p_n **]**	PAR $prio_1$, $startAddr_1$, id_1 ... PAR $prio_n$, startAddr$_n$, id_n PARE *endAddr* $startAddr_1$: P_1 $startAddr_2$: ⋮ $startAddr_n$: P_n *endAddr*: JOIN	Für jeden Thread wird eine PAR benötigt, um die Startadresse, Index und Priorität zu definieren. Das Ende des Threads wird durch die Startadresse des nächsten Threads definiert, mit Ausnahme des letzten Threads, der über PARE definiert wird. Nach der *endAddr* Marke steht das zugehörige join, das am Ende jeder Instanz ausgeführt wird, solange der ||-Operator aktiv ist.
	PRIO *prio*	Ändert die Priorität des aktuellen Threads auf *prio*. Diese Instruktion hat kein Gegenstück in Esterel.

Tabelle 2.2: Definition des *kep* Assemblers

2.2 Einführung KASM

Während der Entwicklung des kep^v wurde eine Möglichkeit gesucht, die Struktur von Esterel zu vereinfachen und die Menge von Befehlen zu verkleinern. Das führte



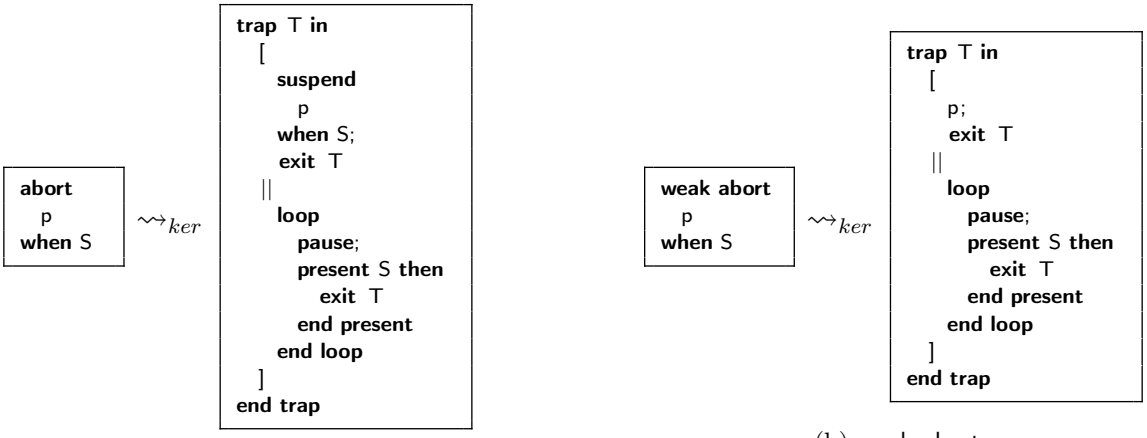
(a) Esterel Code und exemplarische Ausführung

(b) *kep* Assembler

(c) *kep* Ablauf

Abbildung 2.1: Das Esterel ABRO Beispiel

zur Entwicklung der *kep*-Assembler Sprache (KASM). Da viele Ausdrücke in Esterel aus der Komposition von *Kernel-Befehlen* (s. Tabelle 2.1) abgeleitet werden, besteht Esterel im wesentlichen nur aus sehr wenigen Instruktionen. Mit Ausnahme des ||-Operators, werden in KASM werden alle *Kernel-Befehle* direkt ausdrückt. In Tabelle 2.2 ist die Definition der einzelnen KASM-Befehle dargestellt. Um nebenläufige Esterelprogramme auf einem Prozessor mit nur einem Kern ausführen zu können, muss das Programm sequenzialiert werden. Mit der Übersetzung von der Parallelität in Threads durch Li [26] ist dies möglich. Ein Beispiel für diese Übersetzung ist in Abb. 2.1 gegeben. Jeder nebenläufige Block wird in eine par-Instruktion übersetzt, der die Startadresse als Sprungmarke enthält. Die nächste par-Instruktion setzt über ihre Startadresse die Endadresse des vorherigen Befehls. Zum Schluss wird der aufrufende Thread auf die join-Adresse gesetzt. Für diesen Fall hat das KASM Programm die gleiche Semantik wie das ursprüngliche Esterel Programm. Durch die Synchronität von Esterel ist die Übersetzung in vielen Fällen nicht so trivial wie im gezeigten Beispiel. Es können Signalabhängigkeiten zwischen den parallelen Blöcken bestehen, die



(a) abort

(b) weak abort

Abbildung 2.2: Definition von abort aus Kernel Instruktionen

nur durch den Einsatz eines dynamischeren Schedules erfüllt werden können. Daher ist in der Arbeit von Boldt [10] der Compiler `strl2kasm` entwickelt worden, der diese Problematik durch die geeignete Vergabe von Prioritäten löst. Zudem wird mit diesem Compiler die *Worst Case Reaction Time* berechnet, die für die Ausführung von KASM auf dem *kep* mit einem deterministisches Zeitverhalten wichtig ist. Dafür wird ein spezielles Signal (tick_len) zu Beginn jeden Esterel Programms emittiert.

Durch die Auflösung von Esterelblöcken, wie z. B. abort, in eine einzelne Anweisung mit Endadresse, ist bei der Programmierung von KASM ohne den Compiler Vorsicht geboten. Mit dem GOTO Befehl lässt sich beliebig zu jeder Adresse springen [15], der kep^{ε} kann solche fehlerhaften Sprünge nicht erkennen. Ähnliches gilt für die Threads und ihre Prioritäten.

Einige abgeleitete Esterel Befehle werden in der Praxis sehr häufig verwendet. Durch das Auflösen in Kernel Befehle entsteht, wie in Abb. 2.2 zu erkennen, eine vielfache Menge von Befehlen. Um das Anwachsen des KASM-Codes zu verringern, wurden einige wichtige Esterel Befehle in KASM direkt aufgenommen. Zu diesen gehören abort, await und halt. KASM stellt somit eine wichtige Abstraktionsschicht für die Ausführung auf einem reaktiven Prozessor dar und erleichtert die Entwicklung solcher Prozessoren.

2.3 Einführung Esterel v7

Bei der Weiterentwicklung von Esterel zur aktuellen Version v7 wurde insbesondere die Beschreibung von Hardware erleichtert.

Abstraktion

In Esterel v7 können Daten, Schnittstellen und Module voneinander getrennt werden. Dies ist eine wichtige Eigenschaft zur Steigerung der Wiederverwendbarkeit

```
type sig_type  = bool[signal_width];
   map sig_type {
      pre_op[0],
      signal_id [1.. signal_width−1] };
```

Abbildung 2.3: Definition von Datentypen in Esterel v7

von Quellcode. So können jetzt Datenstrukturen zentral definiert werden und von mehreren Modulen genutzt werden. Dies ist auch mit Schnittstellendeklarationen möglich, sogar wenn die Definitionsrichtung sich ändert. Existiert ein Speicher `mem` mit zwei Eingängen und einem Ausgang, dann kann das Modul, das auf diesen Speicher zugreift, das Interface einfach durch Hinzufügen des Schlüsselwortes mirror spiegeln. Module können zudem generisch definiert werden, d. h. ein Modul `add` kann zunächst auf beliebigen Datentypen eine Addition definieren. Erst mit der Instantiierung wird ein Typ übergeben.

Daten

In Esterel v5 sind Daten auf wenige Typen (`integer`,`float` oder `double`) in Esterel selber beschränkt. Nur über die Zielsprache lassen sich eigene Typen definieren. Esterel v7 erweitert die Sprache um selber definierbare Typen in Esterel. Dies wird durch die Einführung von *Arrays* möglich. Die einfachste Arrayform sind *Bitvektoren*, die beliebig lange Folgen von einzelnen Bits sind. Bitvektoren können in einzelne Abschnitte (*Slices*) aufgeteilt werden. In Abb. 2.3 wird die Deklaration gezeigt. Auf die *map* lässt sich dann mit der aus anderen Sprachen bekannten Notation (`?mySignal.signal_id`) zugreifen. Desweiteren lassen sich *signed* und *unsigned* Werte mit beliebigen Definitionsbereichen anlegen. Diese können auch generisch definiert werden. Auf allen *signed* und *unsigned* sind die gängigen arithmetischen Operation definiert, sowie einige Funktionen zur Codierung der Zahlen in Bitvektoren. So ist z. B. `u2onehot` eine Funktion zum Codieren von *unsigned* Werten in die Zahlendarstellung *onehot*.

Darüberhinaus lassen sich Arrays aus allen Typen erstellen, und das auch in beliebigen Dimensionen. Somit wird der Datenfluss auf Signalen entscheidend erweitert, was neben dem Kontrollfluss für die Entwicklung von Hardware wichtig ist.

Datenfluss

Neben dem imperativen Syntax wurde für die Beschreibung von Datenflüssen eine neue Schreibweise eingeführt. Abbildung 2.4 gibt ein Beispiel für die äquivalente Schreibweise von einer imperativen und einer Datenfluss orientierten Ausgabe eines Signals. Diese Art der Programmierung kann effizienter in Hardware übersetzt werden. Im gezeigten Beispiel verkürzt sich die Pfadlänge, da der Test auf dem Signal `I` parallel stattfindet.

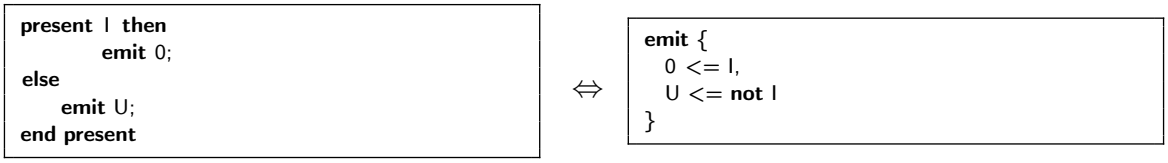


Abbildung 2.4: Datenfluss Syntax in Esterel v7

Sonstige Erweiterungen

Neben einer Datenfluss orientierten Syntax und eigene Datentypen wurden auch Anweisungen zur Steigerung der Effizienz der zu erzeugenden Hardware eingeführt. So können Signale *temp* und *value only* deklariert werden, um unnötige Register zu vermeiden. Auf *temp* Signalen kann nicht auf den pre Zusand zugegriffen werden und bei *value only* besitzt das Signal keinen Zustand, sondern nur Daten.

Zur besseren Verifikation von Programmen sind in die Sprache *Zusicherungen* aufgenommen worden. Bei kritischen Operationen, wie bei dem Indizieren von Arrays, wird dem Programmierer vorgeschrieben Zusicherungen anzugeben. Ansonsten lassen sich weitere Zusicherungen selber definieren, wenn man sicherstellen will, dass bestimmte Abhängigkeiten erfüllt sind.

Zum Testen in nichtdeterministischen Umgebungen können *Orakel* als zufällige Signalquelle angegeben werden. Damit lässt sich das Verhalten in Simulation auf beliebe Eingaben testen.

In der Entwicklung von Hardware spielen Energiespartechniken eine immer größere Rolle. Für diese Techniken müssen Teile der Hardware abgeschaltete werden können, um Energie einzusparen. Dafür benötigt man mehrere treibende Uhren, die sich einzeln schalten lassen. Somit wurde Esterel v7 um ein *Multiclock*-Modell erweitert. Damit können Module mit verschiedenen Uhren betrieben werden, anstatt alle Module synchron zu betreiben. In diesem Zusammenhang wurde auch weak suspend in die Sprache aufgenommen.

2.3.1 Esterel als Hardwarebeschreibungssprache

Mit der Einführung von Esterel v7 verlagert sich der Fokus von Esterel zu einer Hardwarebeschreibungssprache (HDL). Die imperative Syntax für die Beschreibung von Kontrollflüssen bleibt jedoch zusätzlich erhalten. Um zu zeigen wie Hardware aus Esterel erzeugt wird, soll hier (s. Abb. 2.5) das Beispiel `ABRO` aus Abschnitt 2.2 dienen. Auch wenn die Schaltkreissemantik nicht direkt zu erkennen ist, kann man die Übersetzung erahnen: Die Eingänge `A` und `B` führen beide in einen Block (Mitte), das await darstellt. Diese Blöcke werden über ein gemeinsames Register gestartet, und ihre Ausgänge laufen zum *Synchronizer* zusammen, der dann die Ausgabe von `O` steuert.

Da Effizienz eine wichtige Rolle bei der Hardwareerzeugung spielt, ist von Esterel-Technologies ein Leitfaden [2] zur Steigerung der Effizienz herausgegeben worden.

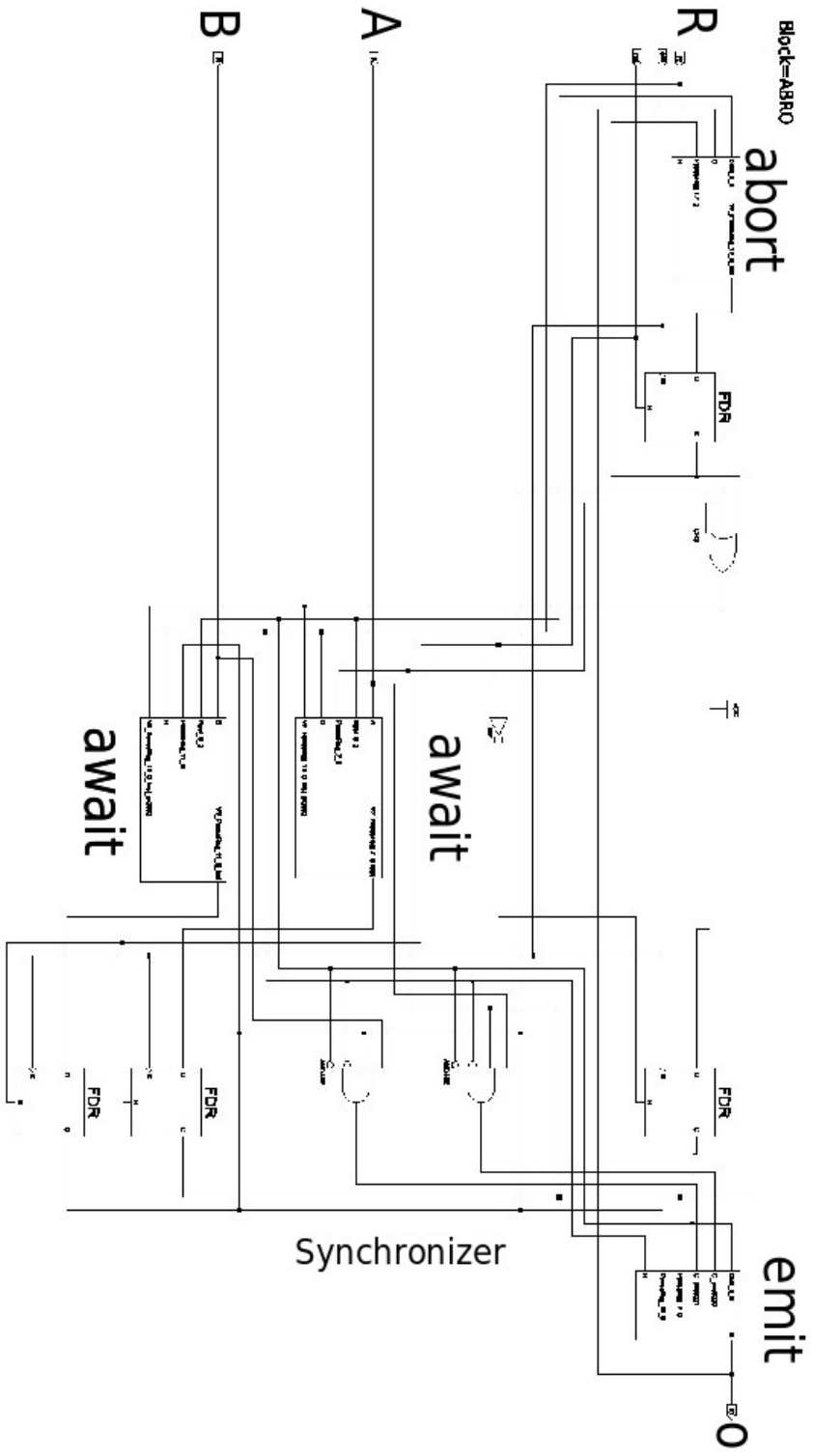


Abbildung 2.5: ABRO in Hardware synthetisiert

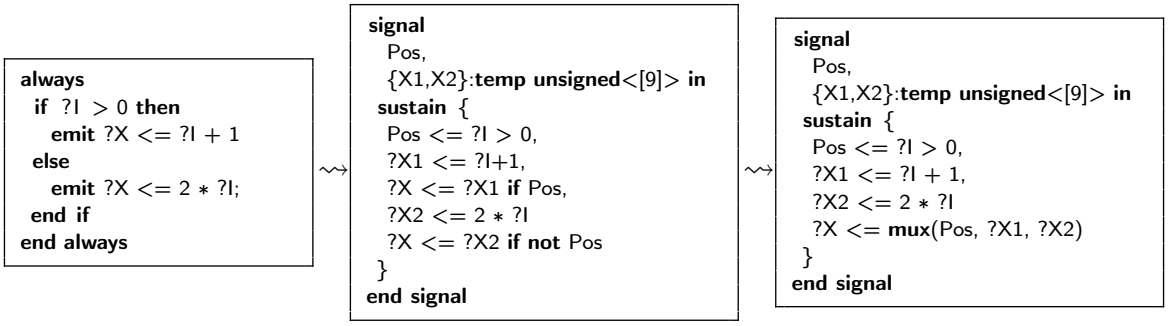


Abbildung 2.6: Verschiedene Stufen der Optimierung von Esterel Ausdrücken

Neben vielen nützlichen Hinweisen wird in dieser Dokumentation klar, das die Effizienz stark von den verwendeten Esterel Ausdrücken abhängt. Als Beispiel ist in Abb. 2.6 die Optimierung einer Konditionalanweisung (`if-then-else`) angegeben. Die erste Version entspricht einer intuitiven Schreibweise, aber jede Ausgabe von `X` hängt von dem vorangehenden *if*-Test ab. In der zweiten Version wird der *if*-Test aus dem Kontrollfluss extrahiert und beide Zweige der Bedingung zunächst parallel lokal emittiert. In Abhängigkeit des lokalen Signals `Pos` wird jetzt der richtige Wert emittiert. Dies verkürzt den Kontrollfluss, ist jedoch weniger lesbar. In einem weiteren Schritt kann noch das doppelte Abfragen des Signals `Pos` über den mux Operator eingespart werden. Diese Form hat fast alle Züge der intuitiven Form verloren. Da diese Optimierungen nicht automatisiert werden, setzt dies ein tiefes Verständnis der generierten Strukturen voraus. Fraglich ist zudem wie bei *SSM* mit diesen Optimierungen umgegangen wird, denn der Esterel-Code wird aus der graphischen Repräsentation automatisch erzeugt.

2.4 *VHDL*

VHDL (Very High Speed Integrated Circuit Hardware Description Language) [13] ist eine Sprache zur Beschreibung von digitalen Systemen. Sie wurde Anfang der 80er Jahre entworfen, um die Entwicklung von Schaltkreisen zu standardisieren und ist heute eine der am weitesten verbreiteten Hardware Beschreibungssprachen. *VHDL* erfüllt im wesentlichen drei Belange an die Schaltkreisentwicklung:

Struktur Beschreibung des Aufbaus und Komposition von Entwürfen.

Funktion Spezifizierung der Arbeitsweise des Entwurfs.

Simulation Testen der Struktur und Funktion nach ihrer Festlegung.

Im Gegensatz zu imperativen Programmiersprachen erfordert die Programmierung in *VHDL* eine Verlagerung der Perspektive in Richtung Hardware. Daher skizziert der geschriebene Code häufig die zu synthetisierende Schaltung. Zwar existieren höhersprachige Elemente, sog. *Prozesse*, jedoch lassen sie sich häufig nicht in Hardware

übersetzten und dienen nur der Simulation und Verifikation. Esterel kann über die Schaltkreissemantik immer in Hardware übersetzt werden, ohne seine imperative Struktur aufgeben zu müssen. Ein weiterer Unterschied zwischen Esterel und *VHDL* ist die Definition des temporalen Verhaltens. In *VHDL* wird für jede Operation eine Verzögerung definiert. Über das Schlüsselwort `after` wird explizit die Dauer der Operation angeben. Häufig ist diese Zeit nicht bekannt und kann somit nicht angegeben werden. Daher wird implizit für jede Operation eine Verzögerung von $\delta > 0$ angenommen. Dabei wird δ als infinitesimal klein angesehen. In der späteren Synthese wird dann das Zeitverhalten analysiert. Somit genügt *VHDL* nicht der *synchronen Hypothese* und lässt damit keine klare Semantik für das Zeitverhalten zu. Damit ist der Entwicklungsprozess für reaktive Systeme zeitaufwendiger, da jede Änderung zunächst undefiniert das Zeitverhalten ändert. Um dies zu analysieren muss zeitaufwendig neu synthetisiert werden.

2.5 Esterel Studio

Esterel Studio ist die integrierte Entwicklungsumgebung für die Hardware und Software Entwicklung mit Esterel v7. Dabei nimmt zunehmend der Entwurf von Hardware den größeren Stellenwert ein. Besonders Controller für Speicher und Eingabegeräte, Protokollimplementierungen und Coprozessor stehen im Fokus von Esterel Studio. Esterel Studio wird von *Esterel EDA Technologies* herausgegeben und liegt zur Zeit in Version 6 vor. Neben der textuellen Unterstützung für Esterel v7 bietet Esterel Studio eine Reihe weiterer Funktionen, die im Folgenden vorgestellt werden sollen.

2.5.1 Zielsprache

Esterel Studio beinhaltet eine Reihe von Compilern für die Übersetzung von Esterel in die Sprachen:

C/C++ ist die klassische Übersetzung von Esterel in Software. Es stehen dabei die zwei Versionen *fast* und *circuit* zu Verfügung. Bei *circuit* wird Esterel entsprechend der Schaltkreissematik übersetzt, während *fast* einen direkteren Weg über den Kontrollflussgraphen des Programms einschlägt [32]. *fast* kann 3 bis 10 mal schnelleren Code erzeugen als *circuit*.

SystemC ist eine Erweiterung von *C* zur Modellierung und Simulation von elektronischen Systemen.

VHDL *Very High Speed Integrated Circuit Hardware Description Language* ist eine der am weitesten verbreiteten Hardwarebeschreibungssprachen.

Verilog ist die zweite wichtige Hardwarebeschreibungssprache. Für Verilog gibt es jedoch einige Einschränkung an die verwendeten Esterel Instruktionen, so kann man z. B. keine zweidimensionalen Arrays in Esterel verwenden.

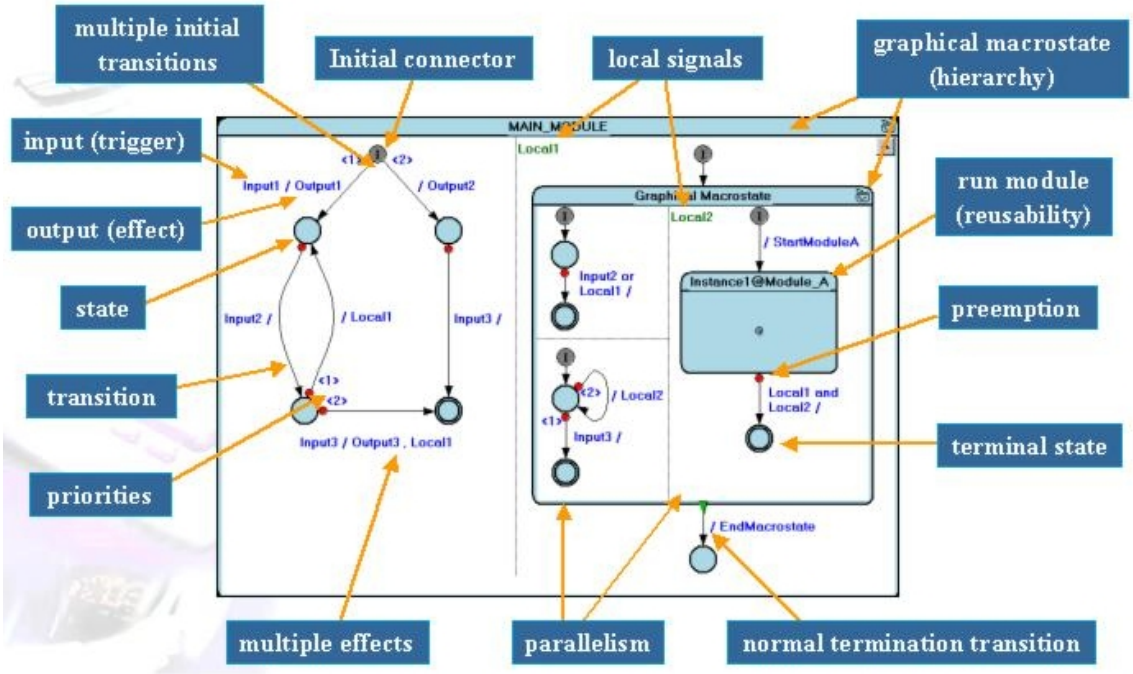


Abbildung 2.7: Verwendung von *SSM* in Esterel Studio (Quelle: [17])

Verification Code ist das Eingabeformat für die in Esterel Studio integrierten Verifikationswerkzeuge.

BLIF Das *Berkeley Logic Interchange Format* beschreibt textuell hierarchische Schaltnetze auf Logikebene. Dieses Format wird für die automatische Optimierung des Codes genutzt.

Mit kleineren Einschränkungen kann aus der gleichen Esterel-Codequelle semantisch äquivalenter Zielcode erstellt werden, der das gleichen Verhalten hat. Bei der in dieser Arbeit vorgestellten Implementierung eines reaktiven Prozessors wurden *C* und *VHDL* als Zielsprachen verwendet.

2.5.2 *SSM*

Esterel Studio integriert Safe State Machines (*SSM*) in dem Entwicklungsprozess von Esterel Projekten. *SSM* ist ein Dialekt von Harels *StateCharts* [24], die von André [1] um eine synchrone Semantik erweitert wurden. In einem *SSM* Diagramm wird das Verhalten durch *Zustände* und *Transitionen* zwischen diesen Zuständen beschrieben. Transitionen sind instantan, während die meisten Zustände, ähnlich wie await oder pause, verzögert sind. Im Gegensatz zu StateCharts sind Transitionen über Hierarchiegrenzen, sog. *Interlevel-Transitionen* hinweg nicht zulässig. Esterel Studio erzeugt aus den *SSM* Esterel-Code, der mitunter sogar lesbar bleibt.

In Abb. 2.7 ist die Verwendung von *SSM* in Esterel Studio kommentiert dargestellt, da auch der kep^{ε} mit Hilfe von *SSM* implementiert wurde.

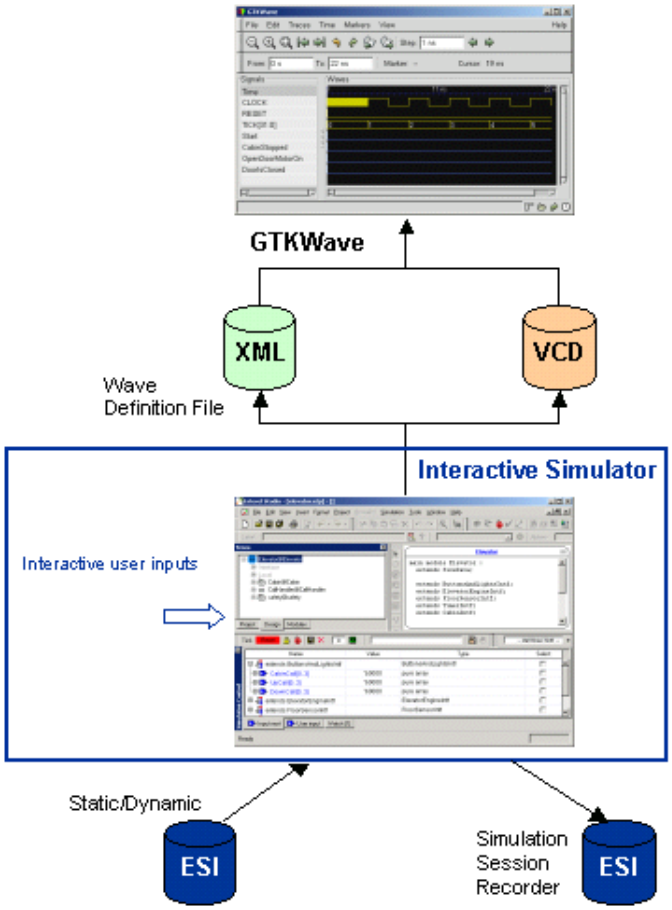


Abbildung 2.8: Benutzung des Simulators in Esterel Studio (Quelle: [17])

2.5.3 Simulator

Eine der interessantesten Funktionen in Esterel Studio ist die integrierte Simulation von Esterel Programmen. Die Programme müssen syntaktisch korrekt und azyklisch sein, um simuliert werden zu können. Im Simulator können dann alle Eingaben des Hauptmoduls frei gesetzt und die Ausgaben beobachtet werden. Zudem lassen sich die Zustände und Daten aller internen Signale einsehen. Während einer Simulation werden die gerade ausgeführten Programmteile farblich markiert. Dies ist vor allem in *SSM* von großem Nutzen, da man hier die aktiven Zustände und die ausgeführten Transitionen deutlich erkennen kann. Die Simulation lässt sich schrittweise, d. h. jede einzelne Instanz, oder kontinuierlich betreiben. Insbesondere bei der kontinuierlichen Simulation ist die Möglichkeit von Haltepunkten sinnvoll, um direkt bis zum Auftreten oder Fehlen von Signalen oder dem Erreichen eines Wertes zu simulieren. Die Ein- und Ausgaben werden zum einen protokolliert, um die gleiche Simulation zu wiederholen, wenn man das Programm verändert. Zum anderen werden die Ausgaben interaktiv in den eingebauten *Logic analyzer* als digitale Wellen gedruckt. Abb. 2.8 ist der Aufbau des Simulators dargestellt. Zusätzlich zum Laden von festen Ein-

gabeszenarien (**.esi*) können diese auch dynamisch aus einem anderen Programm erzeugt werden. Diese Methode wird *Co-Simulation* genannt und soll vor allem für gleichzeitige Ausführung von erzeugten Programmen und Simulation im Quellcode dienen.

Während jeder Simulation werden selbst definierte sowie automatische Zusicherungen geprüft, und man entdeckt damit Fehler, die im erzeugten Code nur schwer zu finden wären.

2.5.4 Verifikationswerkzeuge

Eine der wichtigsten Funktionen von Esterel Studio sind die eingebauten Verifikationswerkzeuge. Diese Werkzeuge sind kaum in anderen Programmen für Hardwaredesign zu finden und dienen dazu, mit formalen Methoden Fehler zu finden, die mit anderen Techniken überhaupt nicht oder nur schwierig zu entdecken gewesen wären. Da früh im Entwicklungsprozess Fehler sichtbar werden, soll sich die Zeit zur Marktreife um 30% verkürzen [35].

Modelchecker

Die formale Überprüfung von Eigenschaften benutzt mathematische Methoden, die zeigen sollen, dass die Eigenschaften des Systems für alle Eingaben an das System erfüllt sind. Insbesondere wird dieser Prozess beim *modelchecking* verwendet und in Esterel Studio vom *design verifier* eingesetzt. Ausgangspunkt für das *modelchecking* sind in Esterel angegebene Zusicherungen oder Programme, die ein externes Verhalten überwachen, sog. *observer*. Bei dem Entwickeln mit Esterel Studio werden auch eine Reihe automatischer Zusicherungen an den geschriebenen Code erstellt. So wird das Lesen von uninitialisierten Variablen oder Signalen, die doppelte Ausgabe von Signalen innerhalb einer logischen Instanz, sowie die Einhaltung von Relationen zwischen Signalen geprüft. Diese können nun ohne Interaktion vom Benutzer automatisch geprüft werden. Bei Verletzungen von einzelnen Eigenschaften wird ein Gegenbeispiel geliefert, mit dem man den Fehler leichter nachvollziehen kann.

Äquivalenztest

Mit Hilfe des Äquivalenztests kann man jederzeit die Äquivalenz zwischen zwei verschiedenen Implementierungen formal zeigen. In Abb. 2.9 ist der Aufbau dieses Tests dargestellt. Die Anforderung für diesen Test an die beiden Module $M1$ und $M2$ ist die Gleichheit ihrer Schnittstelle: genau jedes Signal S aus $M1$ muss in gleicher Richtung und Typ in $M2$ enthalten sein und umgekehrt. Sollte der Test fehlschlagen, erhält man auch hier ein minimales Gegenbeispiel.

Dieser Test ist besonders sinnvoll, wenn in späteren Arbeitsschritten das Design optimiert werden soll, aber das funktionale Verhalten erhalten bleiben muss. Somit lassen sich auch in kritischen Bereichen Änderungen vornehmen, ohne die Stabilität des Gesamtsystems zu gefährden. Darüberhinaus kann man zuerst den Fokus auf das

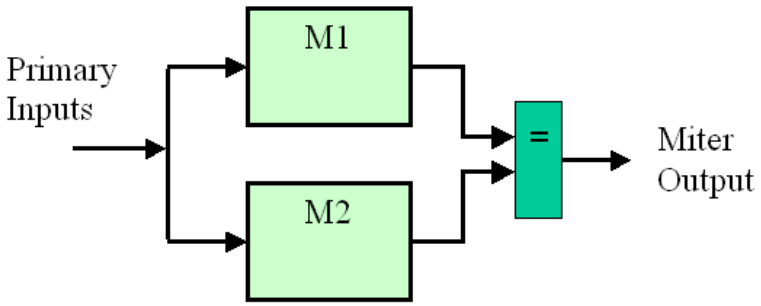


Abbildung 2.9: Schema des Äquivalenztests von zwei Esterel Modulen (Quelle: [17])

korrekte Verhalten setzen und später eine effiziente Implementierung suchen. Dieser Ansatz wurde bei der Implementierung des kep^{ε} mehrfach erfolgreich genutzt.

2.5.5 Zyklische Signale

Esterel Studio ist in der Lage zyklische Signalabhängigkeiten sowohl im Esterel-Code als auch in *SSM* graphisch darzustellen. Die betroffenen Quellcodebereiche werden farblich markiert, und man kann die verursachenden Signale leichter identifizieren. Es können nur azyklische Programme simuliert bzw. in eine Zielsprache übersetzt werden.

2.5.6 Kritische Pfade

Hardware Synthese Werkzeuge wie Xilinx ISE erzeugen nach erfolgreicher Synthese einen Bericht, der unter anderem die maximale Taktrate der Schaltung enthält. Diese Zahl ergibt sich aus dem längsten Pfad in der Schaltung, durch den Informationen fließen können. Darüberhinaus werden die Namen der Register angegeben, über die der längste Pfad verläuft. Diese Register werden meistens bei der HDL Synthese automatisch erzeugt und haben deshalb generische Namen. Esterel Studio bietet nun die Möglichkeit die Namen vom Start und Ende des Pfades in dem Esterel-Projekt anzugeben und den Pfad im Code graphisch anzuzeigen. Damit erhält man die Möglichkeit den *kritischen Pfad* zu optimieren.

2.5.7 Design Abschätzung

Wenn als Zielsprache eine Hardwarebeschreibungssprache gewählt wird, wird ein Bericht über die erzeugte Hardware bei der Codegenerierung erstellt. Dieser Bericht enthält eine Liste über die generierten Register und ihre beeinflussenden Esterel Ausdrücke. Zum einen lassen sich ungenutzte Register erkennen, z. B. wenn ein Bussignal nicht als *temp* deklariert wurde. Zum anderen lässt sich über die Summe der Register eine grobe Schätzung über den Platzverbrauch auf z. B. einem *FPGA* treffen.

Somit kann man früh im Entwurf die Größe der erzeugten Hardware überwachen, auch wenn die Zahl nicht ein exaktes Maß darstellt.

2.5.8 Ausführbare Spezifikation

Esterel Studio bietet die Möglichkeit Spezifikationen zu erstellen und diese ohne Esterel Studio ausführbar auszuliefern. Diese ist vor allem für verteilt arbeitende Firmen und Zulieferer interessant, die ihre eigenen Arbeiten gegen die Spezifikation testen können. Diese Spezifikationen sind deterministisch und frei von Unklarheiten und Missverständnissen, wie sie bei umgangssprachlicher Beschreibung auftreten könnten.

2.5.9 Kritik

Die Markierung von Zyklen im Esterel-Code soll die Identifizierung von zyklischen Signalen vereinfachen. Bei relativ einfachen Abhängigkeiten ist dies auch möglich, jedoch erkennt man diese mit etwas Erfahrung auch selbst. Wird die Abhängigkeit größer, d. h. besteht der Zyklus aus mehreren Signalen, erstreckt sich die Markierung über weite Teile des Quellcodes. Somit bekommt man keinen klaren Überblick, welche Signale für den Zyklus verantwortlich sind. Selbst unbeteiligte Module werden markiert, obwohl in dem aufrufenden Modul nur ein problematisches Signal in der Schnittstellenbeschreibung steht. Hier sollten sich kompaktere Markierungen finden lassen.

Bei der Darstellung *kritischer Pfade* zeigte sich mehrfach ein ähnliches Verhalten wie bei der Zyklusmarkierung: nahezu der gesamte Code war eingefärbt. Somit konnte mit dieser Funktion nichts optimiert werden. Zudem schlug die Anzeige auch mehrfach fehl, obwohl die Registernamen von Esterel Studio erzeugt wurden und die Zuordnung zu dem entsprechenden Esterel Ausdruck leicht zu finden sein sollte.

Der Editor zu Erstellung von *SSM* überlässt das Layout der Graphen fast vollständig dem Zeichner. Dieser legt unter Umständen eigene ästhetische Kriterien fest und beginnt damit den Graphen zu zeichnen. Da auch *SSM* häufig überarbeitet und ergänzt werden, ist bald von der Vorgabe nicht mehr viel zu erkennen. Die in dieser Arbeit abgebildeten *SSM* sind ursprünglich alle mindestens in Leserichtung und Größe definiert gewesen, jedoch ist durch die großen Labels an den Transitionen und wegen hinzugefügten Zuständen das Aussehen zusehends entartet. Ein automatisches Layout mit textueller Editierbarkeit [33] würde hier entscheidende Vorteile bringen.

Die gleichzeitige Ausführung von Programmen und Darstellung in der Simulation ist im Handbuch [17] in zwei Arten beschrieben: über ein Datei (**.esi*) als Schnittstelle und zum anderen als Erweiterung des Codes bei *C* als Zielsprache. Dieser Weg scheint aber nicht mehr unterstützt zu werden, da die Anweisungen aus dem Handbuch nicht zu dem erzeugten *C*-Code passen. So ist man auf die Datei als Schnittstelle angewiesen, was dazu führt, dass der kompilierte Code meist

	Esterel [LOC]	*VHDL* [LOC]	Größe [Slices]	Takt [MHz]
VHDL	-	235	73	214
Esterel	176 (scg2strl)	747	104	192

Tabelle 2.3: Größe und Taktrate von *VHDL* und Esterel `UART` Implementierungen

viel schneller ausgeführt wird als die Simulation es schafft die Eingaben anzuzeigen. Zudem ist die kontinuierliche Simulation auf geöffneten *SSM* sehr langsam.

Für die Implementierung des *kep*$^{\varepsilon}$ in Hardware wurde *VHDL* auf Zielsprache gewählt, da sie keiner Einschränkung an den Esterel-Code unterliegt. In Kapitel 4 sind einige Größen aus der Hardwaresynthese aufgeführt. Ohne im Einzelnen auf die Zahlen einzugehen, lassen sich folgende Aussagen treffen: Die Generierung von Hardware aus Esterel unterliegt im hohen Maße der Art und Weise, wie das Verhalten mit Esterel beschrieben wird. Nach den von Esterel-Technologies [2] veröffentlichen Hinweisen zur effizienten Erzeugung, bleibt dem Anwender häufig verborgen, wann Register erzeugt werden. Ein Beispiel ist in Abschnitt 2.3.1 gegeben worden. Die Optimierungsoptionen aus Esterel Studio können diese Hinweise nicht automatisch umsetzen. Das führt bei naiver Herangehensweise zu ineffizienter Hardware. Aber auch bei Beachtung der meisten Hinweise ist der Platzverbrauch auf einem *FPGA* größer. Ein Beispiel dafür ist der `UART` des *kep*$^{\varepsilon}$ Testdrivers. Die Implementierung wurde von Berry [8] vorgestellt. Diese wurde nur auf den benötigten Anwendungsfall angepasst und ist rund 40% größer und 10% langsamer als eine vergleichbare *VHDL* Implementierung (s.Tabelle 2.3).

Diese schlechteren Werte schlagen sich auch bei größeren Projekten in der Zeit für die Hardwaresynthese nieder. So benötigt man für die Synthese vom *kep*$^{\varepsilon}$ mehrere Stunden bzw. Tage. Damit wird das Testen mit der erzeugten Hardware fast unmöglich.

2.6 *FPGA*

FPGA steht für *Field Programmable Gate Array* und ist ein programmierbarer integrierter Schaltkreis. Aus dem Namen leitet sich auch die Verwendungsmöglichkeit ab, denn FPGAs können ohne größere technische Geräte oder Labore von Jedem beschrieben werden. Ein *FPGA* besteht aus logischen Strukturen, die durch schaltbare Leitungen miteinander zu einer großen Matrix verbunden sind. Jede logische Struktur wird *Configurable Logical Block (CLB)* genannt und besteht aus einem Logikblock mit dem einfache Gatter wie *AND, OR, NOT* oder *XOR* gebildet werden können und einem Speicherelement für die Speicherung der Ergebnisse des Logikblocks. FPGAs vereinen mehrere tausend CLBs, sodass daraus komplexe Schaltungen erzeugt werden können.

Es gibt hauptsächlich fünf Anwendungsgebiete für FPGAs:

(Rapid) Prototyping (schnelle) Entwicklung von Prototypen, die dann zu Test-

zwecken in anderen Systemen zu Einsatz kommen.

Geringe Stückzahl Wenn ein ASIC Entwurf zu teuer würde, kann man aus Kostengründen FPGAs einsetzen, da diese in hohen Stückzahlen produziert werden.

Rekonfigurierbarkeit Im laufenden Betrieb wird der FPGA neu konfiguriert und erledigt eine andere Aufgabenstellung.

HW/SW Co-Design Aufwendige Berechnungen werden auf den FPGA ausgelagert und beschleunigen damit den Prozessor. In der Diplomarbeit von Gädtke [18] konnten mit diesem Verfahren für den *kep*^{v} deutliche Verbesserungen erzielt werden.

Massive Parallelität Mit geeigneten Algorithmen lassen sich Probleme, die sich nur auf Großrechnern lösen lassen, kostengünstig in Hardware realisieren. Das COPACOBANA Projekt [22] ist hierfür ein interessanter Vertreter.

In dieser Arbeit wurde hauptsächlich ein Xilinx *FPGA* der Serie Virtex II pro verwendet. Dieser besteht aus knapp 7000 CLBs. Da die Größe (Anzahl der Eingänge) eines Logikblocks variieren kann, wird die Größe üblicherweise normiert in *slices* ausgedrückt. Für den hier verwendetet FPGA besteht ein CLB aus 2 Slices und damit insgesamt 14000 Slices.

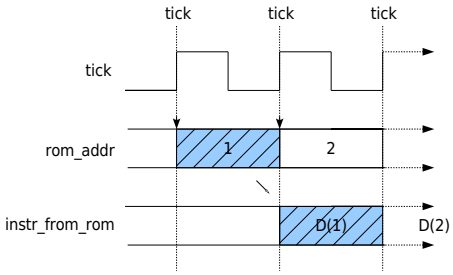
2.6.1 Probleme mit Esterel und *FPGA*

FPGA Hersteller liefern ihre programmierbaren Schaltkreise mit zusätzlichen Eigenschaften aus. So sind z. B. bei der Xilinx Virtex Serie Speicherblöcke integriert, sog. *BlockRam*, die über eine *VHDL* Schnittstelle angesprochen werden können. Diese Speicher sind sehr wichtig, denn wenn man alle Speicher für z. B. Instruktionen mit der Synthese erstellen würde, würde man sehr schnell an die Grenzen des FPGAs stoßen. Desweiteren wird die Synthese unnötig in die Länge gezogen werden, da auch einfache Speicherstrukturen in Schaltkreise übersetzt werden müssten. Auf einem FPGA steht *BlockRam* in festen Blöcken von z. B. $512 \times 36Bit$ oder $4048 \times 4Bit$ zur Verfügung. Meistens benötigt man jedoch eigene Bitweiten und müsste diese Blöcke aufwendig miteinander kombinieren. Daher liegt bei Xilinx ISE ein Programm bei, mit dessen Hilfe sich beliebige Speicher erzeugen lassen. Die Benutzung dieses Generators bringt zudem zwei weitere Vorteile: die Speicherkomposition ist zum einen hoch optimiert und sehr ausgereift, und zum anderen existiert zu dem erzeugten Speicher eine einheitliche Schnittstelle sowie ein Datenblatt mit Operations- und Zeitverhalten. Somit kann man mit Esterel diesen Speicher beschreiben und ihn für die Simulation und Softwareerzeugung nutzen. In Abb. 2.10 ist eine Implementierung des Speichers in Esterel dargestellt. Die Funktionsweise ist einfach: Über den Eingang `rom_addr` wird eine Speicherzelle angesprochen und liegt in der nächsten Instanz am Ausgang `instr_from_rom` an. Für die Schreiboperation existiert zusätzlich der binäre Eingang `wea_instr`, der die anliegenden Daten von `instr_to_rom`

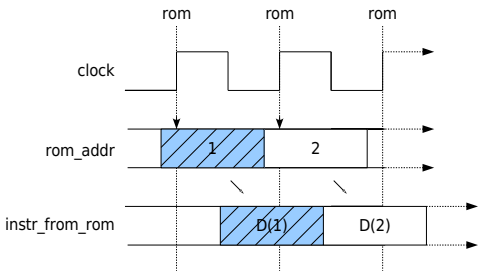
```
module instr_rom:

    extends instr_data;

    input       wea_instr;
    input       rom_addr            : temp value rom_addr_type;
    input       instr_to_rom        : temp value instr_type;
    output      instr_from_rom      : reg value instr_type init '0;

    signal
      // stores to data
        instr_rom[2**rom_addr_width] : value instr_type init '0
    in
      // emit value of the current addr
        loop
            var addr : unsigned<[rom_addr_width]>
            in
                addr:= bin2u(?rom_addr);
                emit next instr_from_rom(?instr_rom[addr])
            end var;
            pause
        end loop
      ||
      // write value to rom
        every wea_instr do
            var addr : temp unsigned<[rom_addr_width]>
            in
                addr:= bin2u(?rom_addr);
                emit instr_rom[addr](?instr_to_rom)
            end var
        end every
    end signal
end module
```

Abbildung 2.10: Implementierung eines Speichers in Esterel

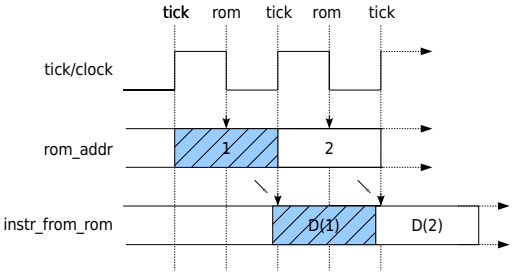
an die aktive Adresse schreibt. Der neue Inhalt des Speichers liegt damit zu Beginn der nächsten Instanz am Ausgang `instr_from_rom` an. Dieses Verhalten ist in dem Datenblatt [37] beschrieben und wird *Write First* genannt, da sich die Ausgabe ohne Veränderung der Adressleitung ändert. Die Simulation aus Esterel Studio und die Softwareübersetzung bereiten für diese Implementierung keine Probleme. Sobald aber der von Xilinx generierte Speicher mit Esterel Studio erzeugter Hardware angesprochen wird, kommt es in unregelmäßigen Abständen zum Stillstand des Schaltkreises. Die erste Vermutung, dass dieses Verhalten im Zusammenhang mit dem Taktsignal und mit der Übernahme der anliegenden Signale in den Speicher steht, führte zur Modifizierung des Speichers auf dem FPGA. In Esterel ist das Taktsignal *tick* mit allen anderen Signalen synchron, d. h. bei steigender Flanke sind die Eingänge zum Speicher entweder *present* oder nicht. Der generierte Speicher ist so spezifiziert, dass bei steigender Flanke des Taktsignals die Daten der Eingänge bereits anliegen müssen. Hier kann es nun zu einem Schnittestellenproblem zwischen den beiden erzeugten Schaltungen kommen: In Abb. 2.11a sind die Signale wie sie Esterel verarbeitet werden abgebildet. Man erkennt, dass die Daten immer zum glei-



(a) Synchrone Implementierung



(b) Spezifikation des Speichers



(c) Umkonfiguration des Speichers auf fallende Flanke und Zugriff vom Esterelprogramm

Abbildung 2.11: Verhalten des Speichers

chen Zeitpunkt anliegen wie das Taktsignal. Für den generierten Speicher (s. Abb. 2.11b) gilt jedoch, dass zu dieser steigenden Flanke Daten schon *stabil* anliegen müssen. Da das Verhalten aus Abb. 2.11a die synchrone Hypothese zugrunde legt, kann man leider nicht immer davon ausgehen, dass die Daten in diesem Moment stabil sind. Um dieses Problem zu lösen, wurde der Speicher auf fallender Flanke konfiguriert. Dadurch liegen die Daten an den Eingängen einen halben Takt eher an als sie gelesen werden (s. Abb. 2.11c). Diese Veränderung führte zur Verbesserung des Verhaltens auf dem FPGA, allerdings traten noch immer sporadisch einige Verklemmungen bei Zugriff auf den Speicher auf. Durch die Verschiebung konnte das Problem auf der Seite des Speichers gelöst werden, jedoch wurde dafür auf der Seite des Esterel Schaltkreises ein neues geschaffen. Die Problematik des Lesens auf instabilen Signalen tritt nun auf dem Ausgangssignal des Speichers auf. Daraufhin wurde die Speicherimplementierung geändert, sodass sie diesem Verhalten entspricht: die Antwort einer Adressänderung wird erst zwei Takte später am Ausgang sichtbar. Somit benötigt man zum Lesen vom Speicher zwei Takte, obwohl letztlich nur einer nötig wäre. Damit traten keine weiteren Abstürze des Schaltkreises auf der Hardware bezüglich der Verwendung von FPGA und Speicher auf. Alternativ hätte man

auch den Speicher mit einem *Handshake Mechanismus* ausrüsten können. Allerdings würde man dann auf Daten warten, was zu nicht deterministischem Zeitverhalten führt und für reaktive Systeme deshalb nicht in Betracht kommt.

3 Implementierung

Im Folgenden wird ausgehend von der Definition des Instruktionssatzes genauer in die Implementierung des Prozessors eingegangen. Besonders die Module, die speziell für einen reaktiven Prozessor wichtig sind, der sog. reaktive Kern, werden eingehend erläutert. Mit dem Testdriver wird eine Schnittstelle zur Validierung und Evaluierung des Prozessors vorgestellt.

3.1 Instruktionssatz

Für den kep^{v} existiert von Li [26] ein Instruktionssatz, der auch als Eingabe für den kep^{ε} genutzt werden könnte. Der kep^{v} wurde über die Dauer seiner Entwicklung um neue Funktionen erweitert, so kam in Version 2 Unterstützung von Signalen mit Daten hinzu und in Version 3 Multithreading. Dabei wuchs auch der Instruktionssatz von anfänglich 24bit auf 36bit an. Durch diese Veränderungen wurde die ursprünglich Struktur einer Instruktion weniger lesbar. Mit der neuen Implementierung des Prozessors in Esterel wurde auch ein neuer Instruktionssatz definiert. Vorteil der neuen Definition ist die aufgeräumtere Struktur. Daraus resultiert bessere Lesbarkeit, Erweiterbarkeit und leichtere Verarbeitung.

Der neue Instruktionssatz ist 40bit lang und bietet mit 8bit Opcode-Index genügend Reserven für etwaige Erweiterungen und Ergänzungen. In Abb. 3.1 ist die Struktur der Parameter des Instruktionssatzes abgebildet.

Es gibt 6 Klassen von Instruktionen mit jeweils unterschiedlicher Länge ihrer Parameter. Damit lassen sich alle Instruktionen bilden, und man erhält einen or-

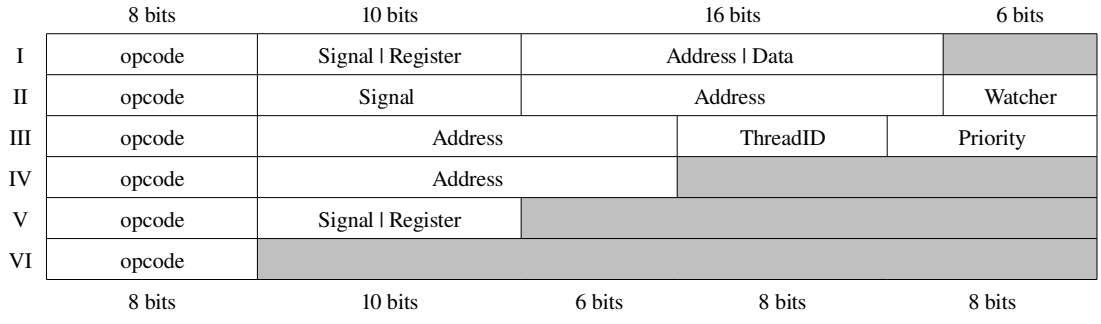


Abbildung 3.1: Struktur des Instruktionssatzes

	fest	variable(bit)	Ersparnis	variable(byte)	Ersparnis
mca200	21564	17875	17%	19688	8,7%
tcint	1849	1403	24%	1585	14,3%
abcd	688	550	20%	621	9,7%
runner	247	182	26%	214	13,3%

Tabelle 3.1: Gegenüberstellung von festem und variablem Befehlssatz

thogonalen Befehlssatz. Ein Auszug des genauen Instruktionssatz wird in Tabelle 3.2 gegeben. Allerdings bleiben bei Klasse I und IV bis VI die grauen Bereiche der Instruktion ungenutzt. Der neue Instruktionssatz ist zudem gut 10% länger ist der vorhandene. Damit stellte sich die Frage, ob man die Vergrößerung des Programmcodes durch eine variable Länge von Befehlen verhindern oder sogar verkleinern kann. Dazu wurden einige bekannte Programme aus der `EstBench` [11] mit variabler und fester Breite übersetzt. In Tabelle 3.1 sind die Ergebnisse darstellt, bei denen sich zeigt, dass man zwischen 17% und 24% der Codegröße einsparen kann. Dies allerdings nur, wenn man bitgenau adressieren würde. Da dies im allgemeinen zu viel größeren Adressen führen würde, wurde die Messung für byteweise Adressierung wiederholt. Die Ersparnis verringert sich auf knapp 9% bis 14%. Wenn man den Mehraufwand der Adressierung und mögliches Nachladen von Parametern gegen die Ersparnis abwägt, kommt man zu dem Schluss, dass eine feste Breite hier vernünftiger ist.

3.1.1 kasmTolst

Im Gegensatz zur klassischen Objektcodeerzeugung gibt es für die Übersetzung von KASM einige Besonderheiten zu beachten: Signale, Register und Variablen sind in KASM noch durch Namen repräsentiert und müssen entsprechend codiert werden. Desweiteren müssen für die abort und suspend Instruktionen Watcher zugeteilt werden. Diese Aufgaben erledigt der Compiler `kasm2lst`, der in `Java` mit Hilfe des Parsergenerator `SableCC` [19] entwickelt worden ist. Daher existiert nun für KASM eine definierte Grammatik, womit etwaige Erweiterungen einfach in den Compiler zu integrieren sind. Auch die Abstimmung mit der Entwicklung des `strl2kasm` Compilers wird dadurch verbessert, da jetzt eine klare Schnittstelle geschaffen wurde. Durch die Verwendung von `Java` wurde zudem die Übersetzung plattformunabhängig.

3.1.2 Übersicht des Befehlssatzes

Der Compiler `kasm2lst` unterstützt die Übersetzung aller KASM Befehle, jedoch werden für den kep^{ε} wegen bestehender Limitierungen (s. Abschnitt 3.5) zur Zeit

Klasse I	Opcode	SignalPre Register	Daten Sprung-Adresse	
	$d_{39} - d_{32}$	$d_{31} - d_{22}$	$d_{21} - d_{06}$	$d_{05} - d_{00}$
EMIT	01000000	SSSSSSSSSP	DDDDDDDDDDDDDDDD	–
SUSTAIN	01001000	SSSSSSSSSP	DDDDDDDDDDDDDDDD	–
LOAD	11010000	RRRRRRRRRR	DDDDDDDDDDDDDDDD	–
PRESENT	00000110	SSSSSSSSSP	AAAAAAAAAAAAAAAA	–

Klasse II	Opcode	SignalPre Register	Sprung-Adresse	Watcherindex
	$d_{39} - d_{32}$	$d_{31} - d_{22}$	$d_{21} - d_{06}$	$d_{05} - d_{00}$
ABORT	10000000	SSSSSSSSSP	AAAAAAAAAAAAAAAA	WWWWWW
ABORTI	10000001	SSSSSSSSSP	AAAAAAAAAAAAAAAA	WWWWWW
WABORT	10000010	SSSSSSSSSP	AAAAAAAAAAAAAAAA	WWWWWW
WABORTI	10000011	SSSSSSSSSP	AAAAAAAAAAAAAAAA	WWWWWW
SUSPEND	10000100	SSSSSSSSSP	AAAAAAAAAAAAAAAA	WWWWWW
SUSPENDI	10000101	SSSSSSSSSP	AAAAAAAAAAAAAAAA	WWWWWW

Klasse III	Opcode	Sprung-Adresse	Treadindex Priorität	Priorität
	$d_{39} - d_{32}$	$d_{31} - d_{16}$	$d_{15} - d_{08}$	$d_{7} - d_{00}$
PAR	01010000	AAAAAAAAAAAAAAAA	0TTTTTTT	pppppppp
PARE	01010001	AAAAAAAAAAAAAAAA	pppppppp	00000000

Klasse IV	Opcode	Sprung-Adresse		
	$d_{39} - d_{32}$	$d_{31} - d_{16}$	$d_{15} - d_{00}$	
GOTO	00000001	AAAAAAAAAAAAAAAA	–	

Klasse IV	Opcode	Sprung-Adresse Priorität		
	$d_{39} - d_{32}$	$d_{31} - d_{22}$	$d_{21} - d_{00}$	
SIGNALIN	00010000	SSSSSSSSSP	–	
AWAIT	00001000	SSSSSSSSSP	–	
AWAITI	00001001	SSSSSSSSSP	–	
JOIN	01010011	00pppppppp	–	

Klasse VI	Opcode			
	$d_{39} - d_{32}$	$d_{31} - d_{00}$		
NOTHING	00000000	–		
HALT	00001011	–		

Tabelle 3.2: *kep*$^{\varepsilon}$ Befehlssatz

nicht alle benötigt. Auch der `strl2kasm` Compiler benutzt nicht alle Instruktionen. Deshalb werden in der Tabelle 3.2 nur die vom Prozessor unterstützen Befehle aufgelistet. Für die Liste aller Definitionen kann man den Compiler mit Option *-opcode <length | name | id | func>* (wahlweise sortiert) starten.

3.2 Aufbau des KEPs

Konzepte klassischer Prozessoren lassen sich auch im *kep*$^{\varepsilon}$ finden. So werden Instruktionen aus dem Speicher gelesen, anschließend decodiert und zur Ausführung gebracht. Aber für die Verarbeitung von reaktiven Instruktionen werden Module benötigt, die die speziellen Eigenschaften in Hardware abbilden. Der sog. *reaktive Kern* beinhaltet die Module, die für die Verarbeitung von Esterel nötig sind. Der *Tick-Manager* stellt das diskrete Zeitverhalten sicher, sodass jeder Esterel Tick

unabhängig von den darin verarbeiten Instruktionen gleich lange dauert. Reaktive Systeme haben typischerweise viele ein- und ausgehende Signale, daher bietet der *kep*$^{\varepsilon}$ für diese Systeme einen speziellen *IO-Controller* über den der Prozessor mit seiner Umgebung kommuniziert. Für Unterbrechungen des Kontrollflusses mit diesen Signalen gibt es mehrere Wächter, mit denen sich einzelne Signale gesondert überwachen lassen, die sog. *Watcher*. Um die nebenläufige Ausführung von mehreren Instruktionen umzusetzen, wurde von Li [26] eine Übersetzung des ‖-Operators mit Hilfe von Threads entwickelt. Dieser Mechanismus kommt auch im *kep*$^{\varepsilon}$ zu Einsatz.

3.2.1 Verarbeitungszyklus

Die Verarbeitung von Befehlen erfolgt im *kep*$^{\varepsilon}$ nach dem Von-Neumann-Zyklus [20] und besteht aus folgenden vier nacheinander ablaufenden Teilschritten.

fetch Der Programmzähler enthält die Adresse der Instruktion, die als nächstes verarbeitet wird. In dem gleichnamigen Modul *fetch* wird ein Befehl an der Stelle des Programmzählers aus dem Instruktionsspeicher gelesen. Die 40bit breite Instruktion wird dem `decoder` übergeben.

decode In dem Modul `decoder` wird die gerade eingelesene Instruktion in ihre Bestandteile zerlegt: der Kopf der Instruktion enthält einen 8bit breiten Index, der zum Identifizieren des Befehls dient. Dahinter folgen meist einige Parameter, die für die Verarbeitung des Befehls notwendig sind. Der Decoder emittiert für jede Instruktion ein Signal, das die Ausführungsschicht veranlasst das entsprechende Modul zu starten.

execute In Abhängigkeit des Signals, das vom Decoder emittiert wurde, wird im Modul `execute` das passende Submodul aktiviert und ausgeführt.

update PC Nach Beendigung der Verarbeitung des dekodierten Befehls wird über das Signal `update_pc` der Programmzähler (Modul `program_counter`) auf die nächste Instruktion gesetzt, im Fall von Kontextwechseln, Abbrüchen oder Sprüngen auf die im Signal enthaltende Adresse.

In Abb. 3.2 sind die Arbeitsschritte exemplarisch für den sequenziellen Fall abgebildet. Dieser Entwurf ist streng sequenziell und verbraucht für jeden Befehl 5 Takte pro Durchlauf (Cycles per instruction, CPI). Dies ist gegenüber modernen Mikroprozessoren langsam, da aber bei der Steuerung von harten Echtzeitsystemen Vorhersagbarkeit Priorität hat, kann man traditionelle Strategien zur Steigerung der Leistung, wie etwa Pipelining und Sprungvorhersage nur begrenzt einsetzen. Einen Kompromiss zwischen sequenzieller Ausführung und teilweiser Beschleunigung wie in *PRET* [16] ist nicht trivial zu finden.

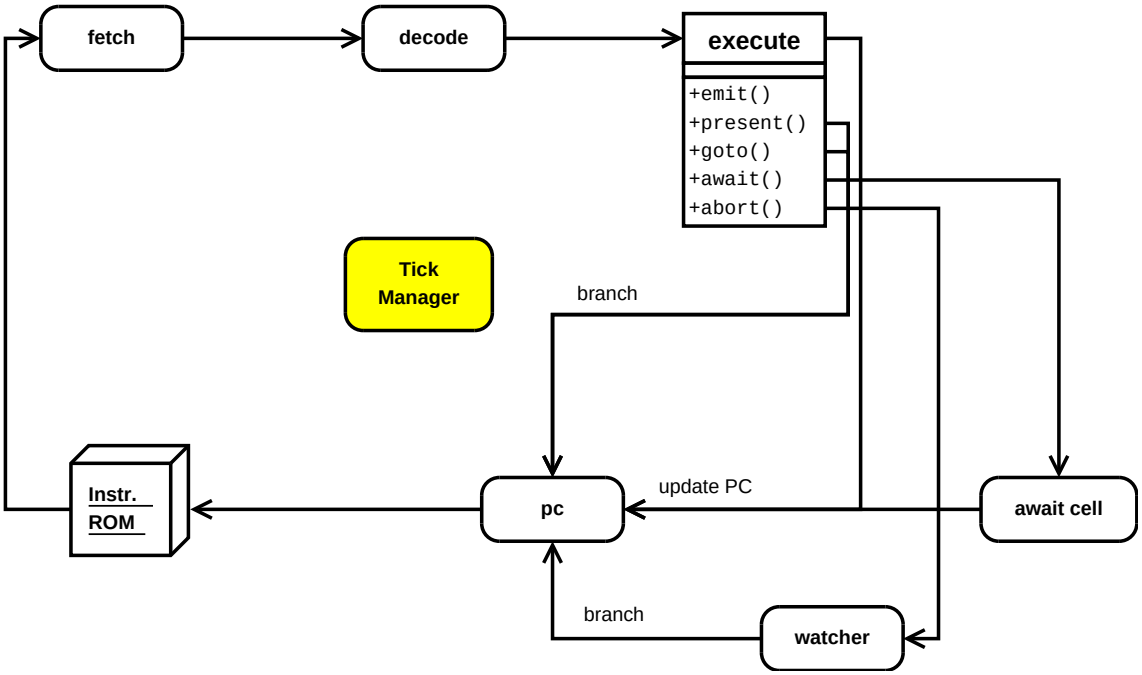


Abbildung 3.2: Von-Neumann-Zyklus des *kep*$^{\varepsilon}$

3.3 Reaktiver Kern

Für den besseren Überblick über die verschiedenen Elemente des reaktiven Kerns, wird in Abb. 3.3 die Interaktion der einzelnen Module schematisch dargestellt.

3.3.1 IO-Controller

Der `io-controller` steuert den Status von Ein- und Ausgangssignalen des Prozessors. Zu Beginn eines logischen Ticks werden die Eingangssignale der Umgebung gelesen und für die Dauer des Ticks in einem Register `cur_signals` gespeichert. Somit wirken sich Änderungen der Zustände der Eingänge während eines Ticks nicht auf das Verhalten aus. Das Register `cur_signals` speichert auch den Status der Ausgangssignale und lokalen Signale. Es besteht somit keine explizite Trennung zwischen Ein- und Ausgängen, womit der Prozessor eine höhere Flexibilität erhält: es können daher je nach Programm mehr Ein- oder Ausgänge definiert werden. Nach Beendigung des Ticks wird dieses Register in das Register `pre_signals` kopiert und zurückgesetzt. Dadurch erhält man zum einen den Zustand für den pre-Operator und zum anderen dient dieses Register als Übergabepunkt für die Ausgänge in die Umgebung. Während eines Ticks kann der Prozessor frei auf den Zustand der Eingänge und der vorherigen (pre) Ein- und Ausgaben zugreifen. Da keine Schreiboperationen auf dem Register `pre_signals` ausgeführt werden, bleibt der Zustand

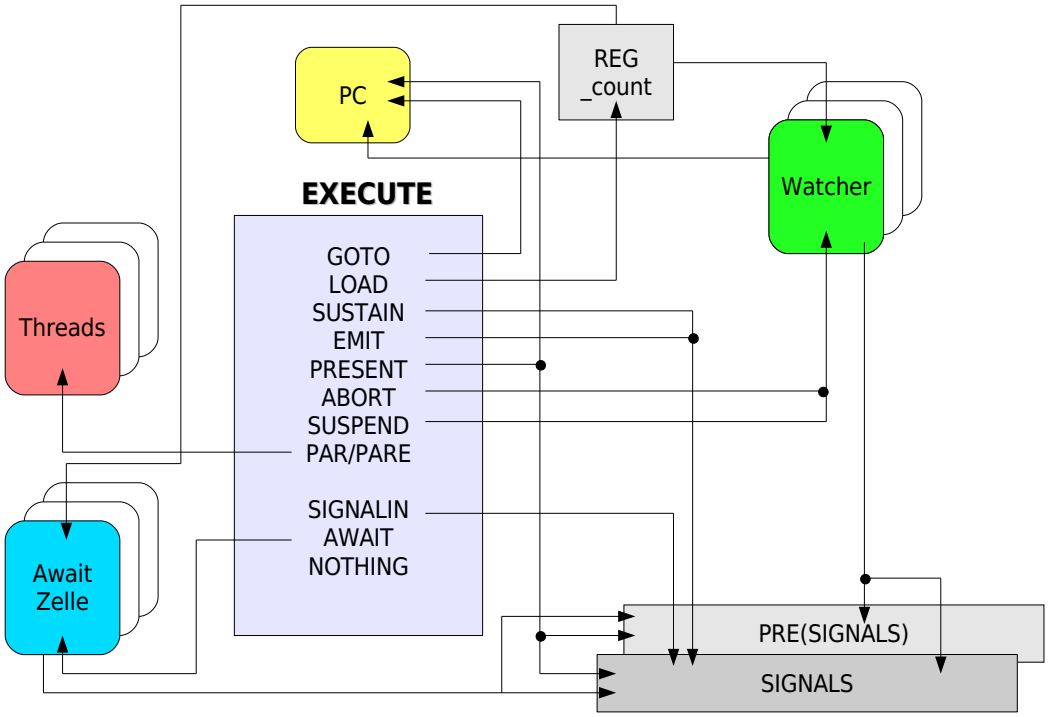


Abbildung 3.3: Übersicht der Elemente des reaktiven Kerns

über die ganze nächste Instanz stabil.

3.3.2 Threads

Eine zentrale Fragestellung ist die Verarbeitung von nebenläufigen Programmen. In Esterel wird Nebenläufigkeit mit dem ||-Operator ausgedrückt. Blöcke, die von diesem Ausdruck getrennt werden, werden gleichzeitig ausgeführt und terminieren, wenn jeder Block terminiert ist. Dies Konzept wird durch die Übersetzung in KASM Threads in äquivalenter Weise abgebildet: Jeder Thread hat einen unabhängigen Programmzähler und eine Endadresse, die den Geltungsbereich angibt. Dieses Konzept ist von Li [26] entworfen worden und in Abb. 3.4 exemplarisch dargestellt. Durch die geeignete Vergabe von Prioritäten vom `strl2kasm` Compiler [10] wird die Ausführung sequenzialisiert. Ein Block von zwei par und einer pare Instruktion erstellt die beiden Threads mit Sprungmarken zu den Startadressen. Die Endadresse ist jeweils die Startadresse aus der vorherigen par Instruktion. In der join Instruktion wartet der Hauptthread auf die Terminierung der beiden abgespalteten Threads.

Damit jeder Thread zur Ausführung gebracht wird, wird ein Scheduler benötigt, der die Threads anhand von Prioritäten ausführt. Desweiteren trägt ein kep^{ε}-Thread die Information, von welchem Thread dieser erzeugt wurde, sowie drei Statusflags:

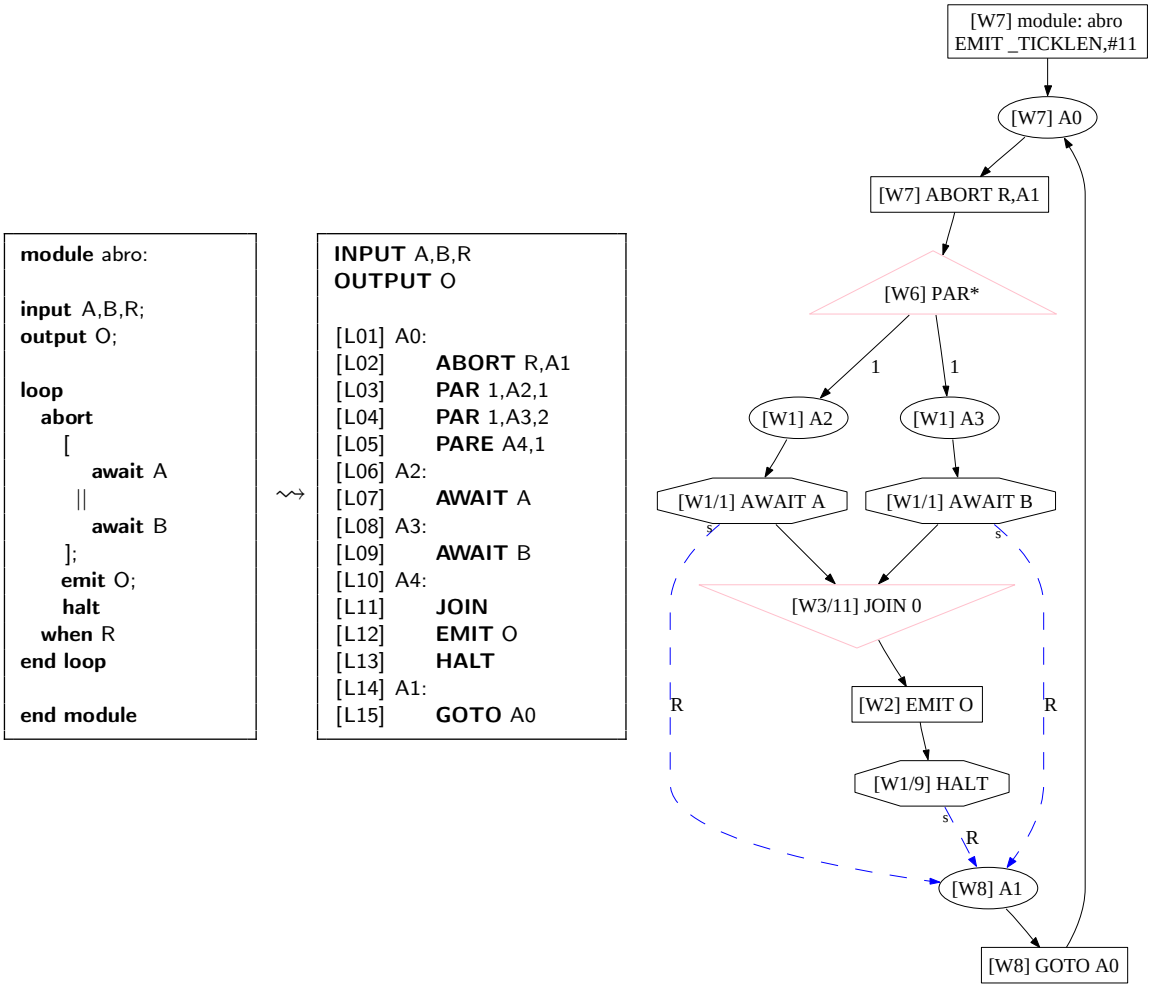


Abbildung 3.4: Übersetzung von || nach KASM Threads und der zugehörige Kontrollflussgraph

running Der Thread ist noch nicht terminiert.

active Der Thread ist in der aktuellen Instanz noch nicht gescheduled worden.

delayed Der Thread hat eine Delayinstruktion (s. Abschnitt 3.3.4) ausgeführt.

Erzeugung

Die *SSM* in Abb. 3.5 beschreibt den Mechanismus zu Erzeugung von Threads. Das Erstellen von kep^{ε} Threads wird in KASM mit einem Block von mehreren par Instruktionen und einer abschließenden pare Instruktion beschrieben. Der Decoder emittiert das Signal `PAR`, dass die Initialisierung der Threads anstößt: Der Scheduler wird durch das interne Signal `thread_init` angehalten, um Inkonsistenzen während der Erzeugung zu vermeiden (s. Abb. 3.5, links). In dem textuellen Makrozustand werden die Parameter *Startadresse* und *Priorität* in das Feld `threads[id]` zur

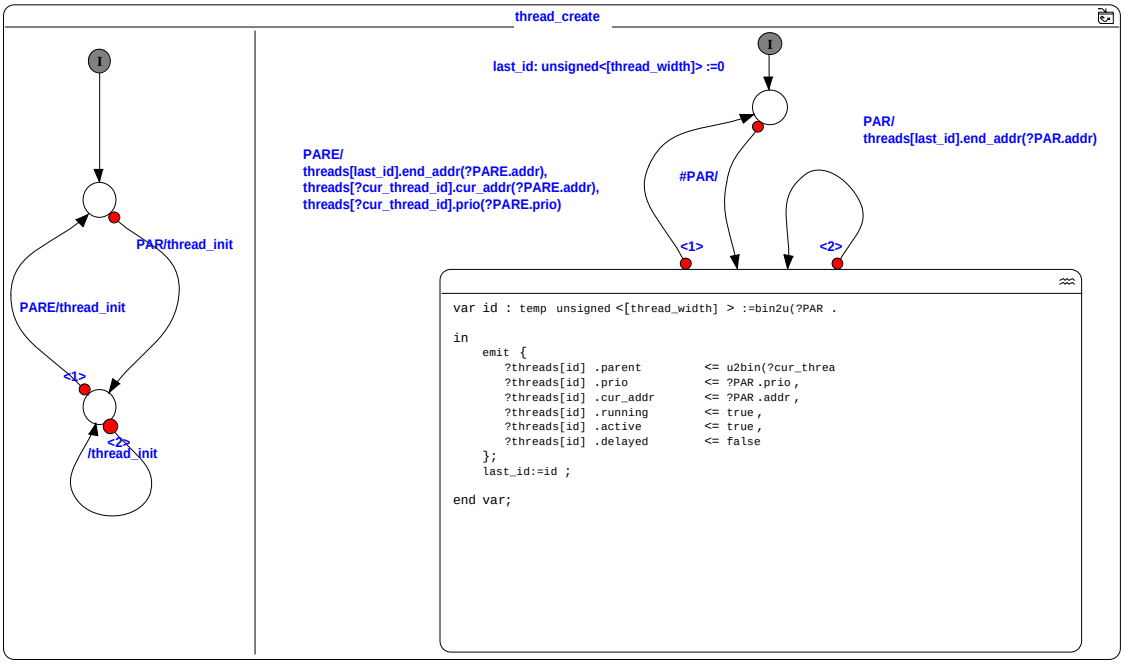


Abbildung 3.5: Zustandsdiagramm zur Erzeugung von Threads

Verwaltung geschrieben. Mit der nächsten par Instruktion wird die *Endadresse* des vorherigen(`last_id`) Threads gesetzt. Dieser Vorgang wiederholt sich bis das Signal `PARE` von Decoder emittiert wird, das die Instruktionskette abschließt. Dabei wird der gerade aktive Thread verändert, indem dessen Programmzähler auf die auf die Adresse des join gesetzt wird. Jeder neue Thread starte mit den gesetzten Flags *active* und *running*.

Ausführung

Die Threads sind nach der Erzeugung *active* und *running*, d. h. es müssen mehrere Threads ausgeführt werden. Der Scheduler `ctsw` entscheidet dynamisch welcher Thread als nächstes ausgeführt wird. Sind mehrere Threads *active*, wird derjenige mit der höchsten Priorität gescheduled. Sollten mehrere Threads die gleiche höchste Priorität haben, bringt der Scheduler denjenigen Thread mit dem höchsten Index zur Ausführung. Durch diesen Algorithmus wird zum einen der Raum der möglichen Prioritäten vergrößert, zum anderen ist die Übersetzung bei einfachen Abhängigkeiten direkt über die Indizes möglich. Hat der Scheduler einen Kontexwechsel festgestellt, wird der Programmzähler auf die aktuelle Adresse des nächsten Thread gesetzt.

Dieses Verhalten ist in zwei Implementierungen umgesetzt worden: Das Modul

`ctsw` enthält eine direkte Beschreibung des Verhaltens durch eine Aneinanderreihung von Abfragen, die die obigen Bedingungen testen. Diese Implementierung ist zwar einfach, hat aber zwei größere Nachteile: Zum einen entsteht bei einer Konfiguration des Prozessors mit vielen Threads eine lange Kette von Bedingungen, die zu sehr ineffizienter Hardware führt, und zum anderen kann dieser Ansatz nicht dynamisch mit verschiedenen Threadkonfigurationen in Esterel implementiert werden. Das letztere lässt sich zwar durch eine Vorverarbeitung automatisieren, aber die Ineffizienz bleibt als schwerwiegendes Manko: Eine Verdopplung der Anzahl der möglichen Threads im kep^{ε} verdoppelt die Pfadlänge auf der Hardwareebene.

Daher existiert eine Variante des Schedulers (`ctsw_fast`), die diesen Nachteil behebt: Bei diesem Ansatz wächst die Pfadlänge nur logarithmisch. Dies wird durch parallele Ausführung der Bedingungen und Speicherung in einer Matrix erreicht. In Abb. 3.3.2 ist die Funktionsweise dargestellt. Initial werden die Indizies der Threads in die erste Spalte geschrieben und paarweise verglichen, welcher von beiden gescheduled werden könnte. Dieser wird dann in die nächste Spalte übertragen. Sollte keiner der Threads gescheduled werken können, wird derjenige Thread mit dem kleineren Index gewählt. Man beachte, dass dieser Vorgang über alle Threads parallel ausgeführt wird und man somit in einer Berechnung die Hälfte aller Threads eliminiert, die nicht gescheduled werden können. Dies wird Spalte für Spalte fortgesetzt, bis nur noch ein Index übrig bleibt. Für n Threads benötigt man also nur $log_2 n$ Schritte um den nächsten Thread zu finden, der gescheduled werden soll. Allerdings benötigt dieser Ansatz $nlog_2 n$ Speicherplatz. Dieser lässt sich durch Optimierung des Ansatzes auf die Hälfte minimieren, indem die erste Spalte bereits mit den Ergebnissen der ersten Berechnung gefüllt wird. Da aber immer noch nur knapp die Hälfte der Matrix benutzt wird, ergibt sich hier noch weiteres Potential für Optimierungen. Esterel erweist sich an dieser Stelle aber als zu unflexibel bei der Deklaration und Instantiierung von Arrays, womit die Optimierungen nicht einfach umzusetzen sind. Um das korrekte Verhalten sicherzustellen wurde der Äquivalenztest von Esterel Studio (s. Abschnitt 2.5.4) zwischen der effizienten und der direkten Implementierung erfolgreich durchgeführt. Dieser Test brachte während der Implementierung einige Programmierfehler zu Tage, die im Regressionstest aufwendig aufzuspüren gewesen wären.

Beide Implementierungen sind instantan und werden während einer Instanz permanent ausgeführt, mit der Ausnahme der Initialisierung von Threads. Wie im linken Teil von Abb. 3.5 zu erkennen, wird während der Erzeugung von Threads das interne Signal `thread_init` emittiert, das den Scheduler anhält, um Inkonsistenzen zu vermeiden. Wenn Kontexwtechsel stattfinden, wird keine zusätzliche Zeit im Sinne von Takten oder Registeroperationen verbraucht, dadurch ist ein Kontextwechsel sehr effizient.

Zu Beginn einer logischen Instanz werden alle noch laufende Threads wieder *aktiv* gesetzt. Desweiteren kann ein Thread über die prio Instruktion seine oder andere Prioritäten ändern und damit dynamisch in das Scheduling eingreifen. Dies ist direkt in dem gleichnamigen Modul umgesetzt.

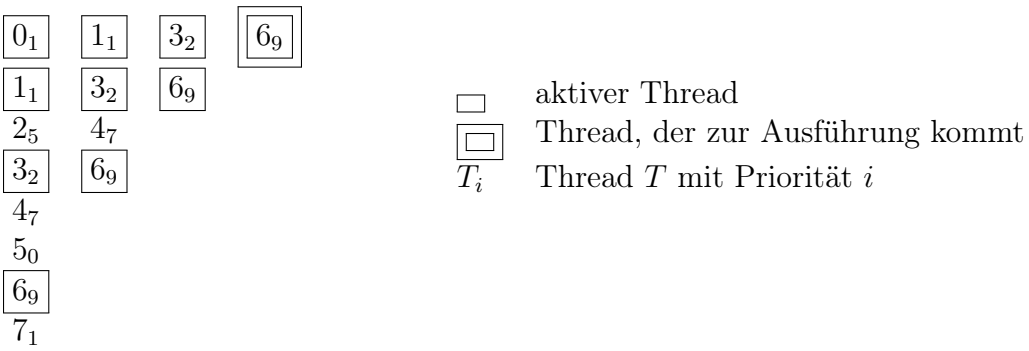


Abbildung 3.6: Effiziente Ermittlung des Kontexwechsels

Terminierung

Das Ende eines Threads kann durch zwei unterschiedliche Weisen zu Stande kommen:

normal Das Modul `term_threads` überwacht den Programmzähler und beendet den aktuellen Thread, wenn dieser seinen Geltungsbereich zu Ende ausgeführt hat, d. h. seine Endadresse abruft.

aborted Wenn ein in der Hierarchie höherer Thread einen Abbruch über eine abort Anweisung auslöst, werden alle Threads, deren Geltungsbereiche verlassen werden, beendet.

Durch die Festlegung der Prioritäten muss gewährleistet werden, dass der aufrufende Thread als letztes gescheduled wird. Dieser Thread führt dann die join Instruktion aus. In dem entsprechenden Modul wird dann geprüft, ob noch Threads laufen, die von diesem Thread erstellt wurden. Wenn es noch Threads gibt, die noch nicht terminiert sind, wartet der aktuelle Thread auf seine Unterthreads, indem er sich *inaktiv* setzt. Wenn keine Threads mehr laufen, wird mit der Verarbeitung der nächsten Adresse begonnen. Dieser Test kann sehr effizient ausgeführt werden, da alle nötigen Vergleiche auf Hardwareebene parallel ausgeführt werden und somit instantan vorliegen.

Im Unterschied zur *kep*^{v} Implementierung in *VHDL* wird das gleiche join nur einmal pro logischer Instanz ausgeführt und nicht auch nach Prioritätsänderungen. Damit verkürzt sich die Anzahl der ausgeführten Instruktionen und damit auch die *WCRT*.

3.3.3 Signalausdrücke

Die Umsetzung der Esterelausdrücke zum Testen und Emittieren von Signalen gehören zu den einfachsten Modulen des reaktiven Kerns.

Emit und Sustain

Das Emittieren von Signalen wird in dem Modul `emit` übernommen. Der im Parameter codierte Index des Signals wird im `cur_signal` Register gesetzt. Damit ist das Signal für die restliche Instanz emittiert. Analog ist das `sustain` Modul implementiert, mit dem Unterschied, dass der Programmzähler nicht verändert wird und somit im nächsten Instruktionszyklus wieder ausgeführt wird. Eine Sustaininstruktion signalisiert auch das Ende des aktuellen Ticks. Bei lokalen Signalen wird mit dem Modul `signalin` dafür gesorgt, dass diese Signale zurückgesetzt werden.

Konditionale Ausdrücke

In Esterel werden konditionale Ausdrücke auf Signalen mit der present Anweisung beschrieben. Der *kep*$^{\varepsilon}$ hat hierfür das gleichnamige Modul, das diesen bedingten Sprung ausführt. Über die Parameter erhält das Modul den Signalindex und prüft, ob das Signal im `cur_signal` Register gesetzt ist. Im pre-Fall wird das Register `pre_signal` geprüft. Bei positivem Test wird der Programmzähler mit der Sprungadresse geladen, die auch als Parameter vorliegt.

KASM spezifische Ausdrücke

Über die load Anweisung können Werte in Register geladen werden. Dies ist hauptsächlich für die Verarbeitung von Datensignalen und deren Zwischenergebnissen nötig. Es gibt jedoch ein spezielles Register `load_count`, das für den Verzögerungszähler in await und abort Instruktionen benötigt wird. Sobald ein await oder abort ausgeführt wird, wird dieser Wert ausgelesen und auf 1 zurückgesetzt. Damit ist es nicht nötig vor jedem await oder abort dieses Register zu laden, wenn nur das einmalige Vorkommen eines Signals benötigt wird.

3.3.4 Delayausdrücke

Zeitliche Abhängigkeiten von Signalen können in Esterel mit der await Anweisung ausdrückt werden. Dies wird von KASM analog mit dem Befehl await umgesetzt[1]. Soll z. B. mit dem Programm nach dem mehrfachen Auftreten eines Signals fortgefahren werden, so muss zuvor das Verzögerungsregister mit load geladen werden. Delayausdrücke markieren auch immer das Ende der Instanz, solange sie nicht immediate sind und die Signalbedingung erfüllt ist.

Sobald eine Delayinstruktion decodiert wurde, werden die Parameter dem Modul `await_cell` übergeben. Das Modul ist als *SSM* implementiert (s. Abb. 3.7), da hier zustandsorientierte Verarbeitung vorherrscht. Da pro Thread maximal eine Delayinstruktion ausgeführt werden kann, existiert für jeden Thread ein Awaitregister. In diesem Register wird der Zustand (`bool await_reg[n].running`) und der Verzögerungszähler des Awaitbefehls (`unsigned<> await_reg[n].count`)

[1] Der pause Befehl wird wie await tick behandelt.

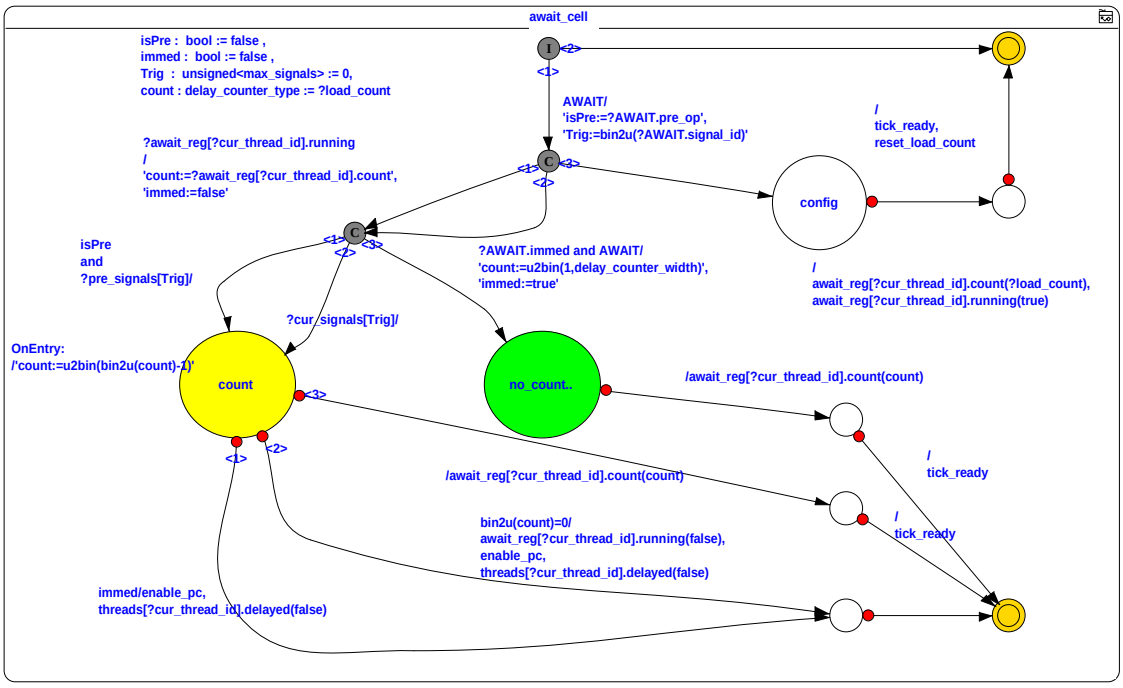


Abbildung 3.7: Zustandsdiagramm der Awaitzelle

gespeichert. Die Ausführung spaltet sich in zwei Fälle auf: Wird an diesem await bereits gewartet (*running*) oder wird das await zu ersten Mal ausgeführt. Im letzteren Fall wird die Awaitzelle konfiguriert, d. h. für den aktuellen Thread wird der Verzögerungszähler aus dem Register `load_count` gelesen und in das Awaitregister geschrieben und der Zustand auf *running* gesetzt. Für diese Instanz ist die Ausführung beendet, und an den `tick_manager` wird das Signal `tick_ready` gesendet, das den aktuellen Thread deaktiviert. Wenn die Awaitzelle bereits für diesen Thread konfiguriert ist, wird der Verzögerungszähler aus dem Register gelesen, und es kann das zu erwartende Signal getestet werden. Dies ist analog zu dem Presentelement aus Abschnitt3.3.3 implementiert. Bei positivem Test wird der Zähler dekrementiert und zurückgeschrieben, und es wird für diesen Thread `tick_ready` gesendet. Ist der Wert des Zählers Null, ist das Await beendet, und der Zustand wird auf `await_reg[n].running=false` gesetzt. In diesem Fall bleibt der Thread noch aktiv, d. h. es wird nicht `tick_ready` gesendet. Im Fall des immediate await wird die Konfiguration ausgelassen, und es verhält sich sofort wie ein normales await nach der Initialisierung. Dies ist möglich, da kein Verzögerungzähler bei immediate await vorkommt.

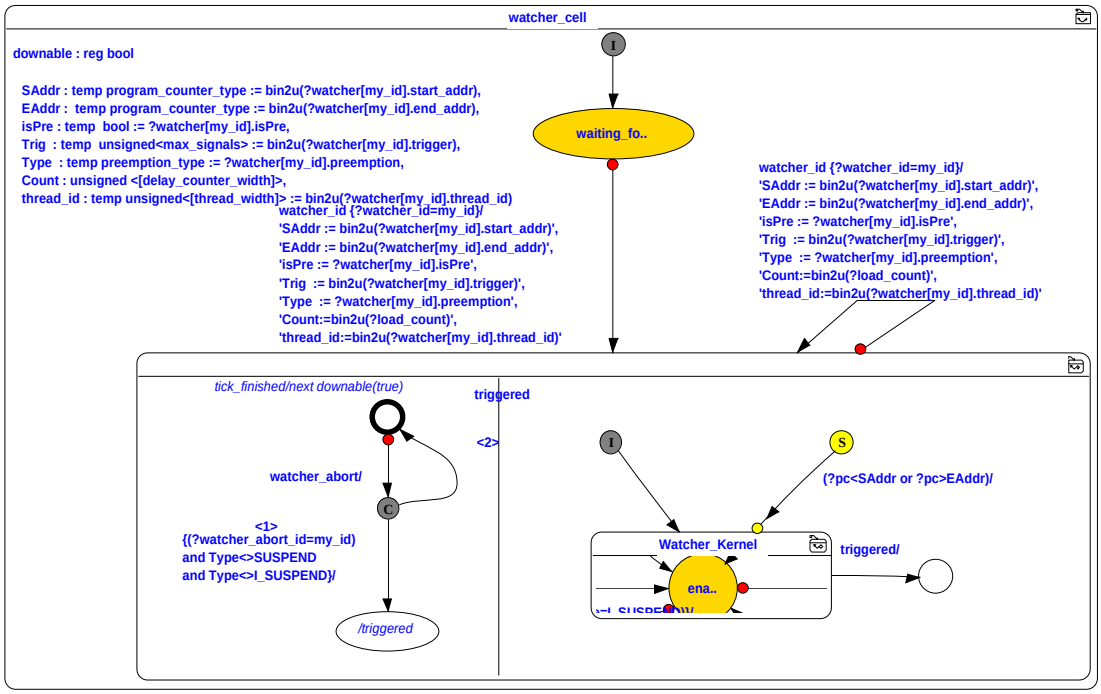


Abbildung 3.8: Oberste Ebene des Watcher

3.3.5 Preemption

Typischerweise werden reaktive Systeme von ihren Eingaben bestimmt und müssen deterministisch auf diese reagieren. Klassische Prozessoren definieren für eingehende Signale Interrupts, die den Kontrollfluss *hart* unterbrechen. Dieser Mechanismus ist für synchrone Sprachen zu unflexibel, da auch der Kontext, in dem das Signal die Unterbrechung auslöst, verloren geht bzw. nicht zur Verfügung steht. Daher haben synchrone Sprache spezielle Signalausdrücke, die Teile des Programms anhalten oder unterbrechen. So finden sich auch in Esterel spezielle Ausdrücke für das Beenden von Programmblöcken (abort) oder das Unterbrechen (suspend), sog. *Preemption* Ausdrücke [4]. Diese Anweisungen können optional mit den Modifikatoren versehen werden:

strong [2] Der Abbruch bzw. die Unterbrechung kann erst mit Beginn der nächsten Instanz erfolgen.

immediate Der Abbruch bzw. die Unterbrechung kann schon in der ersten Instanz erfolgen.

[2] strong abort ist die Vorgabeeinstellung für das abort und wird nicht explizit angegeben.

weak [3] der Abbruch erfolgt erst nach Beendigung der aktuellen Instanz

immediate weak Bei der Komposition der beiden Modifikatoren kann der Block schon in der ersten Instanz abgebrochen werden, aber erst nach Beendigung der aktuellen Instanz.

Wird ein Signal als Auslöser für eine Unterbrechung eines Programmsteils definiert, muss es innerhalb dieses Blockes gesondert überwacht werden. Dafür existiert eine fixe, aber konfigurierbare Anzahl von Wächtern, sog. watcher, die in den Kontrollfluss eingreifen, wenn das Signal anliegt. Dies wird als *feuern* bezeichnet. In Esterel umschließt eine Preemptioninstruktion ihren Wirkungsbereich und ist nur innerhalb von diesem aktiv. In KASM wird hierfür die Endadresse als Parameter übergeben. Sobald eine abort oder suspend Instruktion decodiert wurde, wird über die Parameter eine Watcherzelle konfiguriert. In dem Modul `abort` wird der aktuelle Threadindex, der aktuelle Programmzähler, die Endadresse sowie das auslösende Signal und die Modifikatoren in das Watcherarray gespeichert. Adressiert wird dieses Array sowie die Watcherzellen über einen Index, der vom Opcodekompiler `kasm2lst` erzeugt wird. Zum Schluss wird der Watcher über das interne Signal `watcher_id` aktiviert. Von nun an kann jeder Watcher autonom seinen Zustand überwachen und sich deaktivieren, sobald der Programmzähler sich außerhalb seines Wirkungsbereichs befindet. Wird ein deaktiver Watcher wieder neu konfiguriert, war dieser zuvor *verwaist*, d. h. der Programmzähler hat seinen Wirkungsbereich auf Dauer verlassen bzw. der Watcher ist terminiert. Für diese vielfältigen Zustände empfiehlt sich eine graphische Repräsentation als *SSM* zu verwenden (s. Abb. 3.8). Über die darin enthaltende Hierarchie lässt sich das Verhalten in der Simulation übersichtlich beobachten. In dem Makroknoten `Trigger_Watcher` (s. Abb. 3.8) liegt die Überwachung der Signale. Dabei unterscheidet sich je nach Art des Watchers die Ausführung:

strong abort Nach der Konfiguration befindet sich der Watcher im *enabled* Zustand und wird in der nächsten Instanz mit dem Test auf dem auslösenden Signal beginnen. Dies ist vergleichbar mit dem Signaltest der await-Zelle (vgl. Abschnitt 3.3.4). Über das Flag `downable` wird verhindert, dass in einer Instanz mehrfach der Wert des Zählers vermindert wird. Wenn der Wert des Zählers auf Null fällt, wird in den Zustand *strong* gewechselt. Dabei wird die Art des Watchers sowie die Endadresse dem `watcher_controller` übergeben. Wenn keine anderen Watcher gleichzeitig feuern, ist der Watcher beendet (`triggered`).

weak abort Zunächst verhält sich das weak abort genau wie das strong abort. Fällt der Wert des Zählers auf Null, wird jedoch in dem Zustand *wait* verblieben, bis die Instanz des aktuellen Threads zu Ende gelaufen ist (`thread_ready`). Dann wird auch hier dem `watcher_controller` die Art und Endadresse

[3] Unterbrechungen der Form weak suspend sind in Esterel v5 nicht definiert.

des Watchers übergeben. Wenn keine weiteren Watcher gleichzeitig feuern, ist der Watcher beendet und in den Zustand `triggered` gewechselt.

suspend Bei suspend muss man zwei Fälle unterscheiden: *active*, d. h. es wird der Kontrollfluss angehalten, und *returning*, d. h. der Kontrollfluss kann weiter laufen. Im ersten Fall ist das auslösende Signal aktiv, und es wird von Zustand *enabled* in den Zustand *wait* gewechselt und der aktuelle Programmzähler gespeichert. Hier wird verblieben, bis das Signal wieder erlischt, und es wird dabei dem `watcher_controller` signalisiert, dass ein suspend gerade aktiv ist. Im zweiten Fall war das auslösende Signal in der vorherigen Instanz aktiv, und es wird in den Zustand *suspend* gewechselt. Dabei sendet der Watcher die gleichen Signale zum `watcher_controller` wie bei einem strong abort, jedoch mit dem zuvor gespeicherten Programmzähler als Endadresse, und das Flag `suspend_active` wird wieder frei gegeben. Danach bleibt der Watcher aktiv, um bei dem erneut auftretenden Signal den Kontrollfluss zu unterbrechen.

immediate Modifikator Bei immediate konfigurierten Watchern wird in der gleichen Instanz mit dem Test auf dem auslösenden Signal begonnen, d. h. `downable` ist zu Beginn gesetzt. Im Fall von immediate strong abort wird jedoch der Test bereits während der Konfiguration ausgeführt. Wenn das auslösende Signal aktiv ist, werden direkt dem `watcher_controller` die Parameter übergeben. Im anderen Fall wird das abort wie im nicht immediate Fall konfiguriert.

Die Hauptschwierigkeit bei der Implementierung der Preemptionausdrücke sind nicht die Watcher, sondern die Synchronisation der Watcher, wenn diese verschachtelt auftreten. In Tabelle 3.3 sind alle möglichen Kombinationen enthalten, wie Watcher gleichzeitig in den Kontrollfluss eingreifen und welcher letztlich Priorität erhält. Tiefere Schachtelungsstufen als 2 lassen sich im Allgemeinen auf einen dieser Fälle zurückführen.

Für die Entscheidung, welcher Watcher den Kontrollfluss ändert, müssen alle Watcher der Reihenfolge nach abgefragt werden. Da die naive Implementierung zu ineffizient für einen kep^{ε} mit vielen Watchern ist, wurde hierbei auch eine Implementierung analog dem Prinzip des Threadschedules mit einer Matrix entwickelt. Sobald feststeht, welcher Watcher gewinnt, wird nur noch zwischen *Suspension* und *Abortion* unterschieden. Bei Suspension wird nur das Flag `suspend_active` aktiviert, das den Kontrollfluss für den Geltungsbereich anhält. Wenn das Signal, das für diese Unterbrechung verantwortlich ist, wieder erlischt, erlischt auch das Flag, und der Kontrollfluss kann weiter laufen. Bei Abortion muss jedoch mehr bedacht werden: Alle Threads, die von diesem abort eingeschlossen sind, werden beendet. Das geschieht über einen Vergleich der Watcher Startadresse und der Endadressen der Threads. Zum Schluss erhält der Thread, der diesen Watcher enthält, die Endadresse des Watchers als neuen Programmzähler. Man beachte, dass hierbei nicht explizit neu gescheduled oder der Programmzähler direkt beeinflusst wird, dies wird synchron vom Scheduler erledigt.

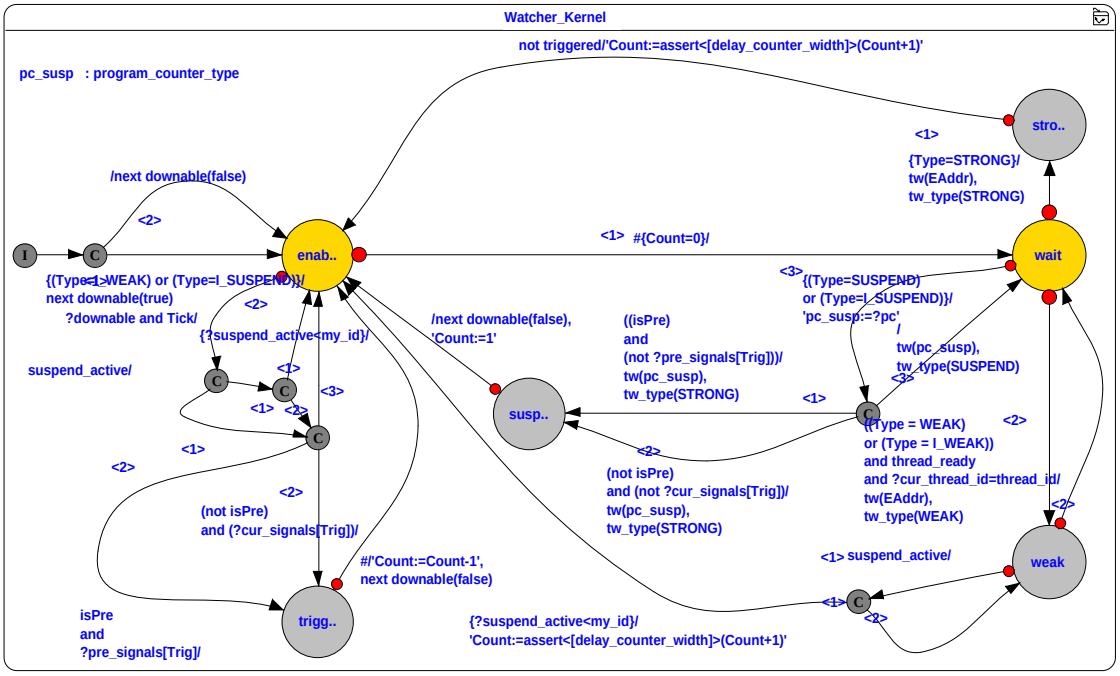


Abbildung 3.9: Kern des Watcher

Das Testen der Signale vom Watcher und die anschließende Verarbeitung benötigt einige Takte. Da die Watcher direkten Einfluss auf den Ablauf des Programms haben, muss der Kontrollfluss zu Beginn jedes Threads verzögert werden. Allerdings gilt dies nur für Threads, die in der aktuellen Instanz zum ersten Mal gescheduled werden. Daher besitzt jeder Thread das Flag *delayed*, das gesetzt wird, wenn der Thread nicht mehr *active* ist.

3.3.6 Tick-Manager

Der `tick_manager` ist eines der wichtigsten Elemente des *kep*$^{\varepsilon}$. Mit dieser Einheit wird das diskrete Zeitverhalten eines Esterel Programms sichergestellt. Mit Hilfe der *WCRT*-Analyse [9] wird ein Wert berechnet, der eine obere Schranke für die zu berechnenden Schritte pro logischem Tick angibt. Typischerweise zu Beginn eines Programms wird dieser Wert über das spezielle Signal `tick_len` an den Prozessor übergeben und mit jeder Abarbeitung einer Instruktion innerhalb eines logischen Ticks dekrementiert. Wenn weniger Instruktionen ausgeführt werden als der Betrag von `tick_len`, wird der Prozessor für die verbleibende Zeit in Leerlauf versetzt. Da jede Instruktion gleich lange dauert, ist dies problemlos möglich. Sollten mehr Instruktionen ausgeführt werden als berechnet, also `tick_len` auf Null fallen, wird

	Preemption	Semantik/Umgebung	Implementierung

1.	**abort**
	  abort
	  p
	  when S
	  q
	when T
		äußeres abort erhält Kontrolle	watcher mit *kleinerem* Index wird gewählt, innerer Watcher ist nicht mehr aktiv

2.	**abort**
	  weak abort
	  p
	  when S
	  q
	when T
			tritt nicht gleichzeitig auf, weil weak abort erst am Ende des Ticks den Watcher verlässt

3.	**abort**
	  suspend
	  p
	  when S
	  q
	when T
		äußeres abort erhält Kontrolle	eine äußere *Abortion* erhält immer Vorrang vor einem laufenden suspend

4.	**weak abort**
	  abort
	  p
	  when S
	  q
	when T
		inneres abort erhält Kontrolle	tritt nicht gleichzeitig auf, weil weak abort erst gegen Ende des Ticks den Watcher verlässt

5.	**weak abort**
	  weak abort
	  p
	  when S
	  q
	when T
		inneres abort erhält Kontrolle	watcher mit *größerem* Index wird gewählt

6.	**weak abort**
	  suspend
	  p
	  when S
	  q
	when T
		S, T emittiert: äußeres abort erhält Kontrolle	eine äußere *Abortion* erhält immer Vorrang vor einem laufenden suspend (vgl. 3)

		$T, pre(S)$ emittiert	*returning* suspend verhält sich wie strong abort und damit analog zu 4.

7.	**suspend**
	  abort
	  p
	  when S
	  q
	when T
		S, T emittiert: suspend wird ausgeführt	verhält sich wie 1., jedoch wird der innere Watcher nicht beendet sondern bleibt erhalten

		$S, pre(T)$ emittiert: q wird ausgeführt	*returning returning* suspend gibt zuerst das Flag `suspend_active` wieder frei, danach können die eingeschlossenen Watcher erst *feuern*

8.	**suspend**
	  weak abort
	  p
	  when S
	  q
	when T
		S, T emittiert: suspend wird ausgeführt	bei laufenden suspend werden die innerern Watcher in ihrer Ausführung unterbrochen

		S,pre (T) emittiert	tritt nicht gleichzeitig auf, weil weak abort erst am Ende des Ticks den Watcher verlässt

9.	**suspend**
	  suspend
	  p
	  when S
	  q
	when T
		S,T emittiert: äußeres suspend aktiv	analog zu 1.

		S,pre (T) emitiert: äußeres suspend aktiv	bei laufenden suspend werden die innerern Watcher in ihrer Ausführung unterbrochen
		$pre(S), T$ emittiert: inneres suspend wird aktiv	*returning* suspend gibt zuerst das Flag `suspend_active` wieder frei, danach können die eingeschlossenen Watcher erst *feuern*
		$pre(S), pre(T)$ emittiert: äußeres suspend kehrt zurück	beide suspend Blöcke können nicht gleichzeitig aktiv sein, daher tritt dieser Fall nicht auf

Tabelle 3.3: Schachtelung von Watcher

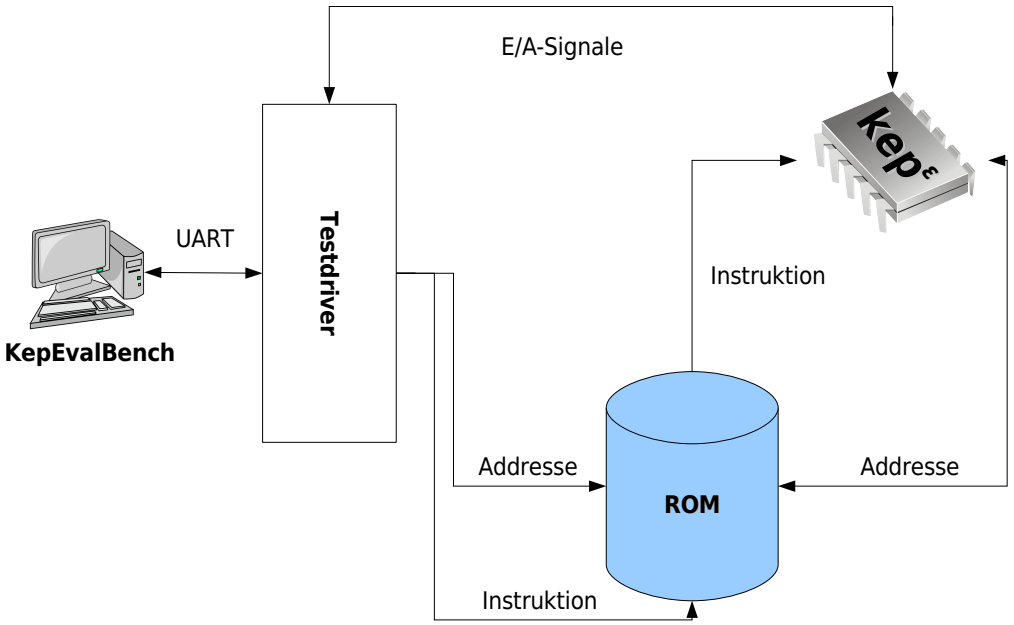


Abbildung 3.10: Architektur von *kep*$^{\varepsilon}$ Testdriver

die Umgebung über die Zeitüberschreitung mit dem Signal `tick_warn` informiert.

Desweiteren stellt der `tick_manager` das Signal `Tick` bereit, mit dem der Prozessor die Verarbeitung beginnt oder fortsetzt. Wenn keine Instruktionen mehr innerhalb der aktuellen Instanz ausgeführt werden müssen, wird der `tick_manager` über das interne Signal `tick_real_ready` informiert, die Instanz zu beenden. Dieses Signal wird vom dem Modul `watch_tick_ready` emittiert, wenn kein Thread mehr in dieser Instanz gescheduled werden kann.

3.4 Testdriver

Der *kep*$^{\varepsilon}$ ist für Prüf- und Demonstrationzwecke mit einem sog. Testdriver verbunden (s. Abb. 3.10). Dieses ebenfalls in Esterel geschriebene Programm enthält Funktionen, mit denen es möglich ist, Daten in den Befehlsspeicher zu laden, Eingaben zu setzen und Ausgaben zu lesen. Diese Schnittstelle kann über ein Protokoll von einem Benutzerprogramm angesteuert werden, der `KepEvalBench` [26]. Mit diesem graphischen Programm kann man Verhalten von Programmen auf dem *kep*$^{\varepsilon}$ überwachen und Eingangssignale frei setzen und deren Ausgaben beobachten. Die Implementierung der Testdriver und seinen Funktionen soll im Folgenden vorgestellt werden.

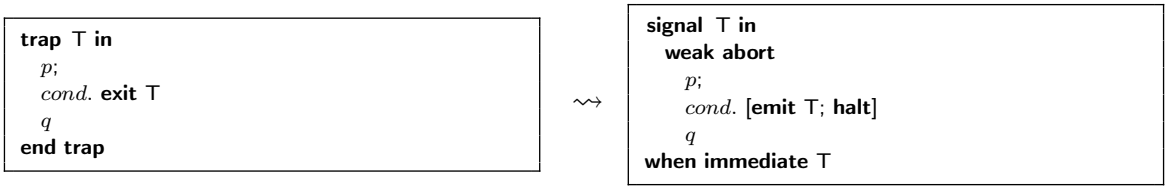


Abbildung 3.11: Ersetzung von Trap durch weak abort

3.4.1 Aufbau

Ähnlich dem aus Abschnitt 3.2.1 bekannten Zyklus verarbeitet der Testdriver Eingaben, die an ihn gestellt werden. Allerdings ist aufgrund des eingeschränkten Eingabeprotokolls die Implementierung wesentlich einfacher. So müssen nur einzelne Zeichen decodiert werden, statt komplexer Instruktionen. Diese Zeichen werden über die serielle Schnittstelle empfangen und Antworten dazu darüber gesendet. Daher existiert im Testdriver das Modul `UART`, das diese Kommunikation in das passende Format wandelt. Die Implementierung stammt von Berry et al. [8] und wurde einigen Optimierungen unterworfen bzw. auf die hier benötige Aufgabe angepasst.

Die wichtigsten Funktionen sind das Übertragen von Programmen in den Instruktionsspeicher, sowie das Setzen und das Auslesen der E/A-Signale. Darüberhinaus lassen sich die Instanzen einzeln starten und somit das Verhalten des Prozessors Schritt für Schritt verfolgen. Dies ist vorallem zur Fehlersuche und zum Verständnis der Programme wichtig. Durch den modularen Aufbau konnte im Rahmen eines Praktikums der Testdriver erweitert werden, sodass nun Programme ohne Interaktion auf dem Prozessor laufen und Eingaben und Ausgaben über externe Quellen wie Schalter und Dioden möglich werden.

3.4.2 Funktionen

3.5 Limitierungen

Dem *kep*$^{\varepsilon}$ fehlt die Unterstützung für Signale mit Wert, den sog. *valued signals*. Dies ist aus Zeitmangel nicht mehr umgesetzt worden, erscheint jedoch relativ einfach. Es genügt eine Standard Arithmetisch-logische Einheit (ALU) für die Operatoren aus Esterel, die dann an einen Wertespeicher angebunden wird. Der Wertespeicher besteht aus zwei Spalten, eine für den aktuellen Wert und eine weitere für den pre-Wert, und verhält sich analog des `io_controllers`. Die Erweiterungen des `decoders` folgen direkt dem bestehenden Muster.

Eine weitere Einschränkung ist die fehlende Unterstützung von trap. Allerdings können einfache Ausnahmebedingungen leicht in eine äquivalente Darstellung ohne Trap überführt werden. In Abb. 3.11 ist für den sequenziellen Fall eine Transformationsregel angegeben, in der die Trap im wesentlich durch weak abort ersetzt wird. Leider lässt sich diese Regel nicht für beliebige Fälle anwenden. In Abb. 3.12 ist

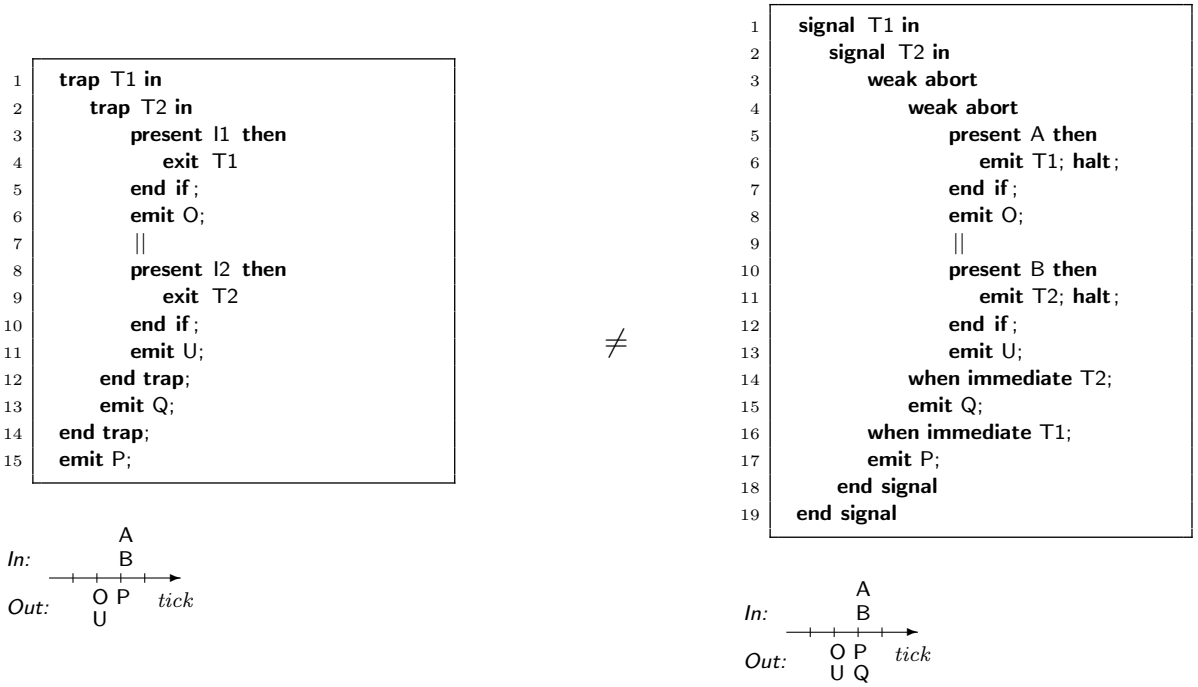


Abbildung 3.12: Parallele Traps können nicht durch weak abort ersetzt werden

ein Gegenbeispiel gegeben. Wenn beide Traps gegangen werden, erhält die äußere Trap Priorität. Bei der weak abort Ersetzung wird jedoch bei Abbruch durch $T1$ und $T2$ noch das emit `Q` aus Zeile 15 ausgeführt. Somit ist eine einfache Transformation geschachtelter Traps nicht möglich, jedoch lässt sich im Einzelfall ein äquivalentes Programm ohne Trap finden.

4 Ergebnisse

Aus der vorgestellten Implementierung wurde VHDL-Code generiert und Hardware synthetisiert. Die folgenden Ergebnisse vergleichen vorallem den kep^{ε} gegen den kep^{υ}. Zudem wird genauer auf den Prozess der Validierung eingegangen und die modellbasierte Hardwareentwicklung bewertet.

4.1 Validierung

Bei Entwicklung von Hardware/Software mit Esterel Studio stehen dem Programmierer Werkzeuge zur formalen Verifikation zu Verfügung. So lassen sich typische Fehler wie Wertebereichsverletzungen von Variablen, doppelte Ausgabe von Signalen während einer Instanz oder das Lesen von uninitialisierten Variablen durch automatische Zusicherungen vermeiden. Zusätzlich können eigene Zusicherungen definiert werden. Für einzelne isolierbare Module wie z. B. den Testdriver und den darin enthaltenen `UART` lassen sich diese Tests auf einem aktuellen Computersystem in einigen Stunden Rechenzeit durchführen. Mit steigender Komplexität des Prozessors ist eine formale Beweisbarkeit unmöglich, da der zu testende Zustandsraum explodiert. Daher wurde hier mit Hilfe von einer genügend großen Menge von Beispielprogrammen die partielle Korrektheit sicherstellt.

Generierung von Eingabeszenarien

Die Testprogramme sind nicht zufällig gewählt, sondern wurden so geschrieben, dass sie eine oder mehrere Eigenschaften des Prozessors testen. So wurden z. B. zum Testen der Verschachtelung von Watcher Testfälle erstellt, die jeweils einen Fall auf der Tabelle 3.3 abdecken. Zudem wurde die `EstBench` [11], eine Sammlung von Esterelprogrammen, in die Testmenge aufgenommen. Man benötigt jedoch nicht nur das Testprogramm (**.strl*), sondern auch möglichst vollständige Eingaben und korrekte Ausgaben dazu. Da die Potenzmenge aller Eingangssignale über eine endlich Anzahl von Instanzen zu Zustandsexplosion führt, kann man nicht einen simplen Eingabegenerator verwenden. Da sich jedes Esterelprogramm in einen Kontrollflussgraphen übersetzen lässt, kann man mit Hilfe eines Überdeckungstests (*Code Coverage*) Eingabeszenarien erstellen, die alle möglichen Zustände des Programmes abdecken. In Abb. 4.1 ist der Ablauf der Generierung dargestellt. Für den Überdeckungstest wurde Esterel Studio v5.5 verwendet, bei dem sowohl Zustände als auch Transitionen in den Test einbezogen werden und auf zwei verschiedene Arten (*short* und *long*) Eingabeszenarien erzeugt. Bei *short* wird die einfachste Sequenz von Eingaben gebildet,

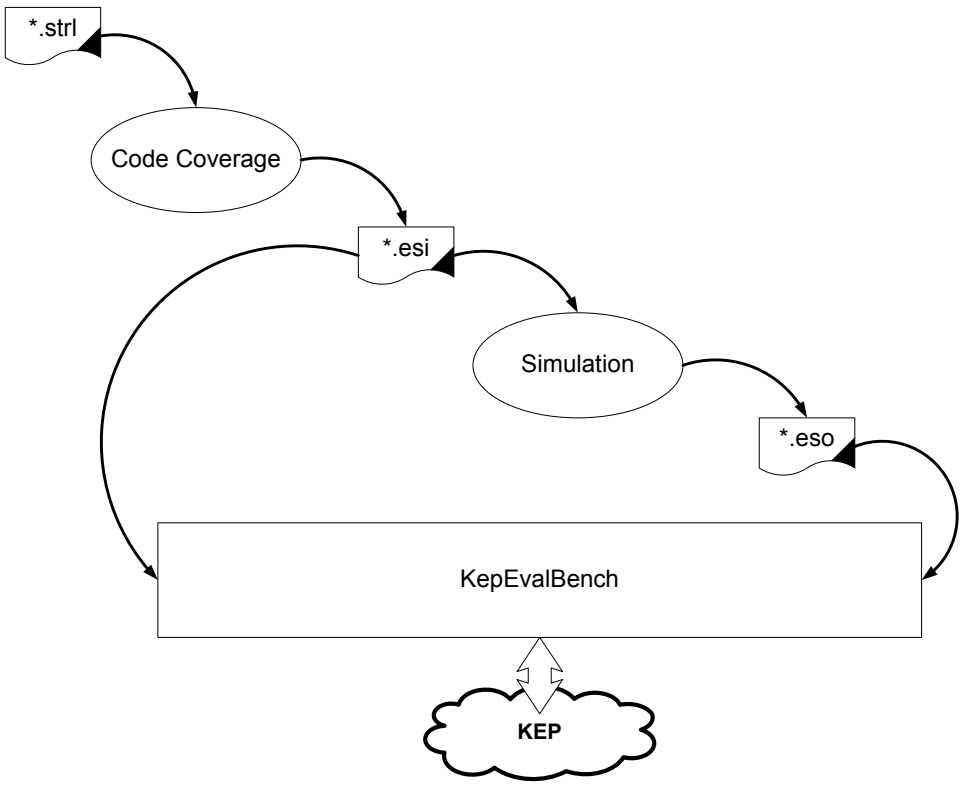


Abbildung 4.1: Erzeugung der Testfälle

um vom Initialzustand zu einem erreichbaren Zustand zu gelangen. Dem gegenüber wird bei *long* eine kurze Sequenz erzeugt, die möglichst viele neue Zustände erreicht. Diese beiden Szenarien werden zusammengefasst (**.esi*) und beinhalten den üblicherweise größten Teil aller sinnvollen Eingabeszenarien. In einigen Fällen kann es zu schlechten Szenarien kommen, wenn die Programme sehr kurz sind und der Kontrollflussgraph nur aus einem Zustand besteht. Dann empfiehlt es sich, zusätzliche await Anweisungen zu Beginn des Programms einzufügen. Mit Hilfe des in Esterel Studio enthaltenen Simulators werden die dazugehörigen Ausgaben erzeugt. Da Esterel Studio als Referenz für die Ausführung von Esterel angesehen werden kann, wird das Testprogramm mit den Eingabeszenarien simuliert. Somit erhält man Referenzausgaben (**.eso*) für das Testprogramm. Mit dieser Methode wurden für 700 Esterel-Programme Szenarien und Referenzausgaben erstellt.

Wegen der bestehenden Limitierungen (vor allem wegen des Fehlens von Signalen mit Daten) kann der kep^{ε} nur gut die Hälfte der 700 Testprogramme ausführen. Da aber das obige Verfahren sich nicht für *valued Signals* eignet, müssen diese Testfälle vorsichtig bewertet werden. Für jede Dateneingabe müsste der gesamte Zustandsraum des Signals in das Szenario aufgenommen werden. Das führt zur Explosion der Anzahl der Eingabeszenarien. Somit müsste die ALU mit formaleren Methoden auf Korrektheit überprüft werden.

Mit der KepEvalBench lässt sich ein solches Eingangsszenario für ein Programm automatisch validieren. Die KepEvalBench sendet nacheinander alle im Szenario enthaltenen Eingaben zum Prozessor und vergleicht die Ausgaben mit denen der Referenz. Über ein Script lässt sich dieses auf beliebig viele Szenarien erweitern und man erhält ein mächtiges Werkzeug für Regressionstests. Während eines solchen Tests wird ein Bericht erstellt, in dem erfolgreiche und fehlgeschlagene Tests und deren möglichen Ursachen aufgelistet werden. Für den *kep*$^{\varepsilon}$ konnten 363 von 375 Tests mit Erfolg ausgeführt werden. Bei den nicht erfolgreich ausgeführten Programmen lag es meist an einer fehlenden oder falschen Festlegung der Prioritäten durch den strl2kasm Compiler. Im ersten Fall vergibt der Compiler für das Programm keine Prioritäten, da es nach der Definition von Boldt [10] als zyklisch erachtet wird. Im anderen Fall liegt es an der unterschiedlicheren Verarbeitung des join vom *kep*$^{\upsilon}$ und *kep*$^{\varepsilon}$ (s. Abschnitt 3.3.2), wodurch die Prioritäten anders gesetzt sein müssen.

softkep$^{\varepsilon}$

Als besonders nützliche Eigenschaft bei der Entwicklung des *kep*$^{\varepsilon}$ in Esterel hat sich die Wahlmöglichkeit bei der Codegenerierung herausgestellt. Die Übersetzung nach *VHDL* führt schon bei kleineren *kep*$^{\varepsilon}$ Konfigurationen zu Synthesezeiten von 1h, bei mittlerer Größe bis zu 10h. Somit lassen sich kleinere Änderungen nicht zügig testen oder dessen Auswirkungen beobachten. Darum wurde schon frühzeitig parallel mit der Übersetzung nach *C* eine Implementierung auf einer x86 Architektur geschaffen. Diese Implementierung wird aus der gleichen Quellcodebasis generiert und unterscheidet sich nur in einigen Schnittstellenbeschreibungen. Die Übersetzung des erzeugten *C*-Codes dauert selbst bei mittlerer Größe nur Sekunden, bei großen und sehr großen Konfigurationen einige Minuten. Der *softkep*$^{\varepsilon}$ lässt sich auch mit der KepEvalBench verbinden, allerdings ist man hier auf die serielle Schnittstelle festgelegt, wodurch Laufzeitmessungen von Programmen auf den unterschiedlichen Implementierungen ihre Aussagekraft verlieren. Jedoch können die gleichen Regressionstests durchgeführt werden wie für die Hardwareimplementierung. Werden Abweichungen vom gewünschten Verhalten eines Testprogramms festgestellt, kann der *softkep*$^{\varepsilon}$ Ausgaben erzeugen, die in den Esterel Studio Simulator geladen werden können. So ist es möglich in Mikroschritten durch die interne Ausführung eines Programms im Prozessor zu laufen und die fehlerhafte Stelle zuverlässig zu finden. Diese Eigenschaft ist ein wichtiger Vorteil gegenüber einem klassischen Hardwareentwurf, bei dem meist keine Softwareübersetzung zu Verfügung steht. Simulatoren für Schaltungen wie z. B. ModelSim können mit dem den Esterel Studio Simulator nicht verglichen werden, da hier nicht der *VHDL* Code selbst simuliert wird, sondern nur die daraus erzeugte Schaltung.

	Watcher	Threads	E/A-Signale	Verzögerungszähler
tiny	4	4	8	16
small	8	8	32	128
medium	16	16	128	256
large	32	32	256	512
huge	64	64	512	4096

Tabelle 4.1: Typische Größen des kep^{ε}

4.2 Vergleich zum kep^{υ}

Der kep^{υ} ist von Li [26] in *VHDL* entwickelt worden. Es sollen hier einige Vergleiche der Implementierung in Esterel gezogen werden, sowie Messergebnisse der erzeugten Hardware. Der kep^{ε} unterstützt aufgrund seines erneuerten Operationscodes bis zu 512 Signale, der kep^{υ} hingegen nur 128. Die Implementierung der Watcher wurde im kep^{υ} in verschiedene Klassen differenziert:

normal Watcher, die für beliebige Verschachtelung von Preemption eingesetzt werden können, auch über Threadgrenzen hinweg.

thread Watcher, die ohne Verschachtelung von Preemption auskommen können und keine Threads einschließen.

local Watcher ohne Verschachtelung von Preemption, jedoch über Threadgrenzen hinweg.

Damit kann der Datenregisterverbrauch minimiert werden, da weniger Informationen über den Kontext in einem Watcher gespeichert werden müssen. Zudem entlastet dieses System den Watcher-Controller, da weniger universelle Watcher benötigt werden. Dies wirkt sich auf die maximale Taktrate aus. Dieses Konzept kann auch vom kep^{ε} in gleicher Weise adaptiert werden.

4.2.1 Laufzeit

Der kep^{υ} benötigt für die Verarbeitung eines Befehls nur 3 Takte, wogegen der kep^{ε} 5 benötigt. Dieser Unterschied ergibt sich zum einen aus der Schnittstellenproblematik aus 2.6.1 und zum anderen aus einer längeren Entwicklungszeit der *VHDL* Implementierung. Durch Optimierung des Prozessors kann sehr wahrscheinlich ein Takt eingespart werden. Leider führen viele Ansätze von Optimierungen zu Zyklen in Esterel Code, die nicht trivial aufzulösen sind. Meist sind die Zykel jedoch *gutartig*, d. h. das Programm ist weiterhin konstruktiv, womit sich der Ansatz von Lukoschus [30] zur automatischen Auflösung von Zyklen anwenden ließe.

Der kep^{ε} führt die join Instruktion nur einmal pro logischem Tick aus, wogegen der kep^{υ} diese Instruktion nach jedem Prioritätswechsel ausführt. Somit ist in diesen

Prozessor Größe	kep^{ε}				kep^{υ}		
	Esterel Studio Register	Slices	Takt Mhz	Synthese(2) (sek.)	Slices	Takt (MHz)	Synthese(2) (sek.)
tiny	471	1411	33,99	148	1461	41,46	77
small	1049	3316	24,64	416	1938	40,72	108
medium	2417	7037	18,72	1065	4005	40,74	220
large(1)	5026	16245	14,48	13462	6267	38,3	372
huge(1)	11131	-	-	-	7609	38,08	610

(1) kep^{υ} mit nur 128 E/A-Signalen (2) Die Synthese wurde auf einem *Intel Xeon 5140 @ 2.33GHz* System mit 8GB Hauptspeicher durchgeführt

Tabelle 4.2: Vergleich zwischen kep^{ε} und kep^{υ}

Fällen die *WCRT* für den kep^{ε} kürzer und damit die Laufzeit aller Instruktionen dieses Programms schneller. Ansonsten haben kep^{ε} und kep^{υ} das gleiche Laufzeitverhalten und Aussagen des kep^{υ} gelten unter Berücksichtigung der Unterschiede auch für den kep^{ε}.

4.2.2 Skalierbarkeit

Sowohl der kep^{υ} als auch der kep^{υ} lassen sich relativ frei konfigurieren, z. B. in der Anzahl von Threads oder Watchern. Einige typische Größen mit deren Details sind in Tabelle 4.1 dargestellt. In Tabelle 4.2 sind für verschiedene Größen des kep^{ε} und kep^{υ} Messwerte aus der Codegenerierung und Hardware Synthese eingetragen. In Esterel Studio v5.5 stehen zwei Wege bei der *VHDL* Erzeugung offen: *monolithisch* und *modular*. Bei monolithischer Generierung wird das gesamte Esterel Projekt in ein *VHDL* Modul übersetzt, während bei modularer Übersetzung für jedes geeignete Esterel Modul ein *VHDL* Modul erzeugt wird. Der Vorteil modularer Übersetzung besteht in der besseren automatischen Optimierbarkeit von großen Projekten. Der Wert *Esterel Studio Register* wird bei der monolithischen *VHDL* Generierung ausgegeben, da dieser nur hier verfügbar ist[1]. Diese Zahl gibt eine synthetische Größe an wie viele Register das Design benötigt. Es werden sowohl Daten- als auch Kontrollflussregister angegeben. Der kep^{ε} wurde dahingehend optimiert, keine unnötigen Datenregister zu verwenden, indem nach Möglichkeit alle Signale als temp definiert wurden. Dadurch werden keine Register für den pre Zustand angelegt. Die Kontrollflussregister lassen sich jedoch nicht so einfach optimieren, da diese implizit durch viele Esterel Ausdrücke erstellt werden. Letztere haben üblicherweise eine Größe von 1 und fallen kaum ins Gewicht. Für *Esterel Studio Register* ergibt sich ein linearer Anstieg mit Faktor 2 bei wachsenden kep^{ε} (vgl. Abb.4.2). Dieser Verhalten entspricht der Erwartung, da die Größe des Prozessors von Schritt zu Schritt verdoppelt wurde. Idealerweiser sollte sich dieses Verhalten auch auf in dem Platzverbrauch auf

[1]Esterel Studio v6 bietet auch bei modularer Codeerzeugung Registerinformationen an.

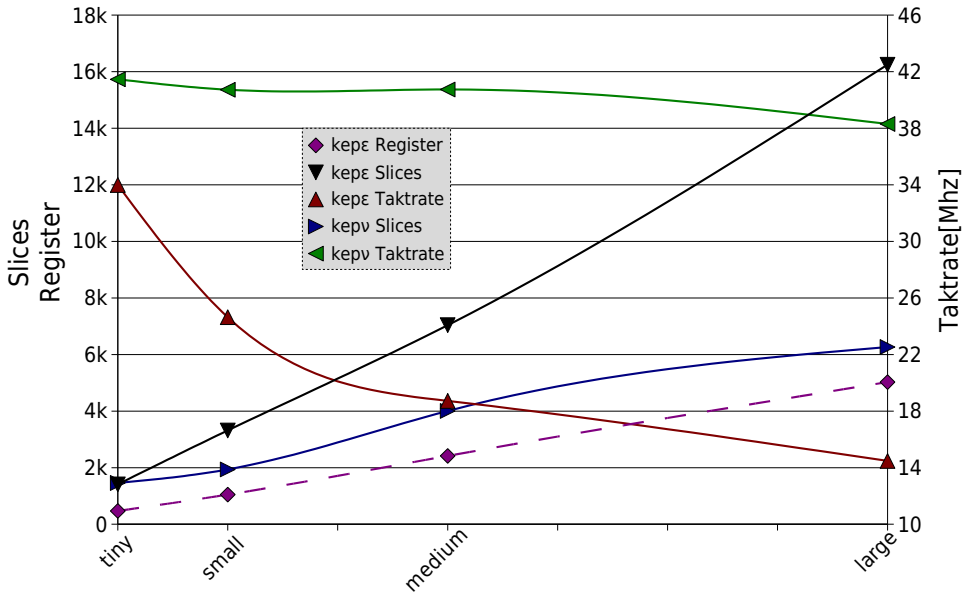


Abbildung 4.2: Entwicklung der Größe des kep^{ε} und kep^{υ}

dem *FPGA* wiederspiegeln. Aber hier ergibt sich ein anderes Bild: die Größe wächst zwar linear, aber mit Faktor 2,3. Damit scheint der generierte Code schlechte Eigenschaften bei der Synthese zu besitzen. Somit muss hier noch tiefer in die Analyse des generierten Esterel Codes, sowie daraus generierten *VHDL* und dessen Synthese eingestiegen werden, um die Ursache des abweichenden Verhalten zu erklären.

Im direkten Vergleich zwischen kep^{ε} und kep^{υ} schneidet der kep^{ε} sowohl im Platzverbrauch als auch bei der Geschwindigkeit schlechter ab. Anders als bei der Softwareentwicklung hängen Platzverbrauch und Geschwindigkeit häufig voneinander ab. Das erklärt sich dadurch, dass die Pfade in großen Schaltkreisen länger werden können und damit die Taktrate heruntergesetzt werden muss. Somit ist die Ursache der schlechteren Werte zum einen in dem Problem der schlechteren Skalierung der Hardware zu suchen, zum anderen steckt im kep^{υ} eine längere Entwicklungszeit. Bei der Erzeugung der Größe *large* brach die modulare *VHDL* Generierung bei einer Nutzung von knapp 1,4GB Hauptspeicher wegen eines Speicherzugriffsfehlers ab. Ohnehin hätte die Synthese dieser Größe vermutlich Tage gedauert, da die Synthesezeit exponentiell wächst. Hier lässt sich ein klarer Unterschied im Skalierungsverhalten erkennen, denn die Synthesezeit für den kep^{υ} steigt nur linear.

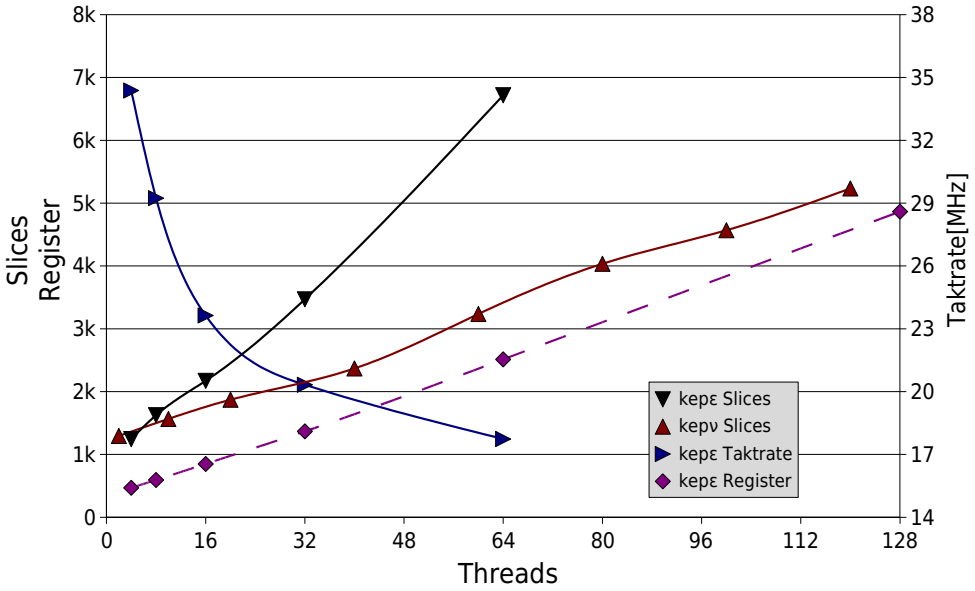


Abbildung 4.3: Größes des kep^{ε} und kep^{υ} in Abhängigkeit von der Anzahl der Threads

Nebenläufigkeit

Desweiteren wurde untersucht, wie sich die Größe und Leistung des kep^{ε} entwickelten, wenn nur die Anzahl der Threads erhöht wird. Diese Messung ist in gleicher Weise in [26] mit dem kep^{υ} unternommen worden, womit sich die Ergebnisse vergleichen lassen. In Abb. 4.3 wurde neben dem Platzverbrauch auf dem *FPGA* des kep^{ε} und kep^{υ} auch die Anzahl der Register des kep^{ε} nach der monolithischen *VHDL* Generierung aufgetragen, da die kep^{ε} Synthese für höhere Anzahl von Threads nicht in tolerierbarer Zeit terminierte (Abbruch nach 10 Tagen). Das Ergebnis zeigt ein ähnliches Verhalten wie im obigen Experiment: Die Anzahl der Register verhält sich erwartungsgemäß, der Platzverbrauch auf dem *FPGA* steigt steiler. Somit bestätigt sich das vorherige Ergebnis, womit sich Teile der Implementierung als mögliche Quelle der schlechteren Werte ausschließen lassen. So wurden z. B. die Anzahl der Watcher in diesem Versuch nicht verändert und gehen als konstanter Faktor in die Messwerte ein. Damit ist ein weiteres Indiz gefunden, dass die Übersetzung von Esterel in Hardware nicht in jedem Fall gut funktioniert. Die Taktrate fällt für den kep^{ε} logarithmisch und verhält sich damit wie durch den Algorithmus im Scheduler zu erwarten war.

Modellbasierte Entwicklung von Hardware schlägt sich im direkten Vergleich etwas

schlechter. Durch den höheren Abstraktiongrad des Modells wird aber der Aufwand für Änderungen stark verringert. Zudem ist die Entwicklungsgeschwindigkeit höher, und einige Module können noch durch effizientere Varianten ersetzt werden. Somit kann die vorliegende Version als Referenz für weitere Äquivalenztests dienen.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde die Implementierung eines *Application-Specific-Instruction-Processor* (ASIP) zur Verarbeitung von Esterel in Esterel vorgestellt. Dabei wurde Esterel als Hardwarebeschreibungssprache eingehend untersucht. Der kep^{ε} ist vollständig in Esterel implementiert und unterstützt mit Ausnahme von trap alle *Kernel Befehle* von Esterel, die für die Ausführung von *pure Signals* nötig sind. Zusätzlich wurde eine Reihe weiterer Befehle implementiert, um die Ausführung zu beschleunigen, da sie nicht in Kernel Befehle umgewandelt werden müssen. Zu diesen gehört vor allem abort, womit der kep^{ε} auch ohne trap volle Unterstützung für alle Arten von Preemption hat.

Aus diesem Modell wurde VHDL generiert und daraus anschließend Hardware synthetisiert. Die Synthese erwies sich als sehr zeitaufwendig und konnte nicht für jede Änderung am Modell durchgeführt werden. Mit wachsender Projektkomplexität konnte alleine über den Simulator der Entwicklungsumgebung der Prozessor nicht mehr getestet werden. Daher wurde auch *C*-Code aus dem Modell erzeugt, der sehr viel schneller compiliert werden konnte. Dieser Software-Prozessor, der sog. $softkep^{\varepsilon}$, konnte zur Validierung mit Hilfe von Regressionstests genutzt werden. Diese Tests sind auch für die Hardwareübersetzung gültig, da die Codegenerierung aus Esterel semantisch äquivalenten Zielcode erzeugt.

Die Implementierung in Esterel wurde gegen die Implementierung in VHDL von Li [26] verglichen. Die erzeugte Hardware ist im direkten Vergleich zum kep^{v} nicht so effizient und verbraucht mehr logische Gatter. Dies ist zum einen ein Effekt der automatischen Codegenerierung, die im Einzelfall hinter handgeschriebenen Code zurückfällt. Zum anderen stellt die hier vorgestellte Implementierung einen Prototypen dar und wurde nur an besonders kritischen Stellen wie z. B. dem Scheduler optimiert. Weitere Optimierungen lassen sich durch die formale Verifikation der Äquivalenz ohne Verlust des bisherigen Verhaltens durchführen. Dabei wird jedoch die Lesbarkeit des Modells leiden, da Optimierung bedeutet, den imperativen Stil von Esterel zu vermeiden und sich an den neuen Datenfluss Syntax zu halten.

Probleme bei der Implementierung traten bei dem Ansprechen von Speicherblöcken auf dem FPGA mit Esterel auf. Es zeigt sich, dass das synchrone Verhalten nur innerhalb von Esterel garantiert werden kann. Mischungen von VHDL Kernen und Esterel funktionieren nur sicher, wenn Handshake-Protokolle benutzt werden. Dadurch leidet jedoch das deterministische Verhalten.

Der Testdriver für den kep^{ε} ist modular aufgebaut und wurde bereits im Rahmen eines Praktikums erweitert, um dem Prozessor in realen Umgebungen einzusetzen. So ist bereits geplant den Prozessor exemplarisch für einfache Steuerungen von Motoren und LEDs zu nutzen. Aber auch ohne Testdriver könnte der Prozessor als

reaktive Komponente für ein *System on a Chip* (SoC) eingesetzt werden. Das Verhalten des darauf ausgeführten Programms würde auf dem *kep*$^{\varepsilon}$ ausgeführt, wogegen datenintensive Berechnungen auf einen anderen Chip ausgelagert würden. Somit ist die fehlende Unterstützung für *valued Signals* kein gravierender Mangel, da dafür bereits genügend Lösungen existieren.

Der *softkep*$^{\varepsilon}$ ließe sich als Software-Prozessor für Esterel nutzen und könnte Anwendung in der Lehre finden, vor allem wenn zusätzlich eine Evaluierungs- und Visualisierungssoftware geschrieben würde. Damit könnte man die Ausführung von Esterel ähnlich wie in Esterel Studio simulieren.

6 Literaturverzeichnis

[1] Charles André. SyncCharts: A Visual Representation of Reactive Behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996. http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf.

[2] L. Arditi, G. Berry, J. Dormoy, L. Blanc, S. Granier, and M. Perreaut. Generating efficient hardware with Esterel v7 and Esterel Studio, April 2005.

[3] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003.

[4] Gérard Berry. Preemption in concurrent systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 72–93, London, UK, 1993. Springer-Verlag. ISBN 3-540-57529-4.

[5] Gérard Berry. *The Constructive Semantics of Pure Esterel.* Draft Book, 1999. ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps.

[6] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *Lecture Notes in Computer Science (LNCS)*, pages 389–448. Springer-Verlag, 1984. ISBN 3-540-15670-4.

[7] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992. http://citeseer.nj.nec.com/berry92esterel.html.

[8] Gérard Berry, Michael Kishinevsky, and Satnam Singh. System Level Design and Verification Using a Synchronous Language. In *International Conference on Computer-Aided Design (ICCAD '03)*, page 433, 2003.

[9] Marian Boldt. Worst-case reaction time analysis for the KEP3. Study thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2007. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-st.pdf.

[10] Marian Boldt. A compiler for the Kiel Esterel Processor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, November 2007. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-dt.pdf.

[11] CEC. Estbench Esterel Benchmark Suite. http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz.

[12] C.M.Edmund Chow, Joyce S.Y.Tong, M.W.Sajeewa Dayaratne, Partha S Roop, and Zoran Salcic. RePIC - A New Processor Architecture Supporting Direct Esterel Execution. School of Engineering Report No. 612, University of Auckland, 2004.

[13] Pong P. Chu. *RTL HARDWARE DESIGN USING VHDL Coding for Efficiency, Portability,and Scalability.* A Wiley-Interscience publication, 2006.

[14] M. W. Sajeewa Dayaratne, Partha S. Roop, and Zoran Salcic. Direct Execution of Esterel Using Reactive Microprocessors. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, 2005.

[15] Edsger W. Dijkstra. GOTO considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[16] Stephen Edwards and Edward A. Lee. The case for the precision timed (pret) machine. Technical report no. ucb/eecs-2006-149, EECS Department, University of California, Berkeley, November 2006. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-149.pdf.

[17] *Esterel Studio User Manual.* Esterel Technologies, 6.0 edition, December 2007.

[18] Sascha Gädtke. Hardware/Software Co-Design für einen Reaktiven Prozessor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2007.

[19] Etienne Gagnon. SableCC: Java parser generator. http://sablecc.org/.

[20] M. D. Godfrey and D. F. Hendry. The computer as von neumann planned it. *IEEE Annals of the History of Computing*, 15(1), 1993.

[21] Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9), September 1991.

[22] T. Güneysu, C. Paar, J. Pelzl, G. Pfeiffer, M. Schimmler, and C. Schleiffer. Parallel Computing with Low-Cost FPGAs: A Framework for COPACOBANA. In *ParaFPGA Symposium LNI 2007, Jülich, Germany*, September 2007.

[23] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. http://citeseer.nj.nec.com/halbwachs91synchronous.html.

[24] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[25] Kayhan Küçükçakar. An ASIP design methodology for embedded systems. In *Proceedings of the seventh international workshop on Hardware/software codesign (CODES'99)*, pages 17–21, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-132-1. http://doi.acm.org/10.1145/301177.301190.

[26] Xin Li. *The Kiel Esterel Processor: A Multi-Threaded Reactive Processor*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, July 2007. http://eldiss.uni-kiel.de/macau/receive/dissertation_diss_00002198.

[27] Xin Li and Reinhard von Hanxleden. The Kiel Esterel Processor - a semi-custom, configurable reactive processor. In Stephen A. Edwards, Nicolas Halbwachs, Reinhard v. Hanxleden, and Thomas Stauner, editors, *Synchronous Programming - SYNCHRON'04*, number 04491 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/159.

[28] Xin Li and Reinhard von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Dijon, France, April 23–27 2006.

[29] Xin Li, Jan Lukoschus, Marian Boldt, Michael Harder, and Reinhard von Hanxleden. An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 225–236, New York, NY, USA, September 2005. ACM Press. ISBN 1-59593-149-X. doi: http://doi.acm.org/10.1145/1086297.1086327.

[30] Jan Lukoschus. *Removing Cycles in Esterel Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, July 2006. http://eldiss.uni-kiel.de/macau/receive/dissertation_diss_2015.

[31] Sumio Morioka. CQPIC: PIC micro computer free soft IP. http://www02.so-net.ne.jp/~morioka/cqpic.htm.

[32] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007. ISBN 0387706267.

[33] Steffen Prochnow and Reinhard von Hanxleden. Statechart Development Beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA, October 2007.

[34] Zoran A. Salcic, Partha S. Roop, Morteza Biglari-Abhari, and Abbas Bigdeli. REFLIX: A processor core for reactive embedded applications. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *Proceedings of the 12th International Conference on Filed Programmable Logic and Applications (FPL-02)*, volume 2438 of *Lecture Notes in Computer Science*, pages 945–945, Montpellier, France, September 2002. Springer. ISBN 3-540-44108-5.

[35] Esterel EDA Technologies SAS. New esterel studio product and eda market web site, 2008. `http://www.esterel-eda.com/`.

[36] Johann Weber. Entwurf eines risc-prozessors mit synchronen sprachen. Master's thesis, technische Universität Kaiserslautern, 2004.

[37] *Single-Port Block Memory Core v6.2.* Xilinx Inc, 4.0 edition, March 2005.

[38] *Xilinx ISE 9.2i Software Manuals.* Xilinx Inc, 2007. `http://toolbox.xilinx.com/docsan/xilinx92/books/manuals.pdf`.

A Anleitungen

In den folgenden Anleitungen wird erläutert, wie der *kep*$^{\varepsilon}$ und *softkep*$^{\varepsilon}$ erzeugt werden. Es wird davon ausgegangen, dass folgende Programme installiert sind: gnumake, perl, gcc, Esterel Studio v6, xilinx ISE v.9.2. Desweiteren ist für den Betrieb der `KepEvalBench` sicherzustelllen, das die serielle Schnittstelle RS232 auf COM1/ttyS0 mit 115200baud, 8 Databits, ohne Paritätsprüfung und ohne Flusskontrolle eingestellt ist.

A.1 *kep*$^{\varepsilon}$ Konfiguration

Der *kep*$^{\varepsilon}$ kann in vielen Parametern konfiguriert werden. In Tabelle A.1 sind einige Konfigurationen vorgegeben. Diese Größen können bei der Erzeugung von Code direkt angegeben werden und erstellen den *kep*$^{\varepsilon}$ entsprechend der Einträge in der Tabelle. Falls andere Konfigurationen als die Vordefinierten benötigt werden, muss die Datei `sripts/kep.cfg` in dem Block custom angepasst werden. Watcher und Threads müssen Zweierpotenzen sein. Ansonsten kann man alle Werte frei wählen. In Tabelle A.2 ist eine Beispielkonfiguration angegeben.

A.1.1 *VHDL* Generierung

Bestimmung der Taktrate

Je nach Größe der Konfigurationen sollte die effektive Taktrate 3-15MHz betragen. Da die meisten FPGAs keine Uhr in diesem Bereich anbieten, muss aus den angebotenen Uhren eine passende abgeleitet werden. Da aber auch der `UART` im *Testdriver* von dieser Uhr abhängt, muss diese Ableitung zueinander passen. Über das Script `sripts/calc_clk.pl` werden die nötigen Parameter berechnet. Das Script wird

	Watcher	Threads	E/A-Signale	Verzz.	Instr. ROM	Tracebuffer
tiny	4	4	8	16	64	32
small	8	8	32	128	128	64
medium	16	16	128	256	512	256
large	32	32	256	512	1024	512
huge	64	64	512	4096	4096	1024

Tabelle A.1: Typische Größen des *kep*$^{\varepsilon}$

Parameter	Beispielwert	Bedeutung
size	"custom"	keine Bedeutung
WATCHER	32	Anzahl gleichzeitiger Watcher (zweier Potenz)
THREADS	32	Anzahl gleichzeitiger Threads (zweier Potenz)
DATA	128	max. Größe des Verzögerungszählers (load) für await und abort
SIGNALS	64	Anzahl der aller Signale (inkl. lokale und tick_len)
INSTR	512	Größe des Befehlsspeichers (Anzahl Instruktionen)
TRACE	128	Größe des Ausführungsspeichers (min. tick_len)

Tabelle A.2: Beispielkonfiguration zur Parametrisierung des *kep*$^{\varepsilon}$

mit `perl calc_clk.pl -clock=`x gestartet, wobei x die Uhr des FPGAs in KHz ist. Es wird eine Liste mit allen gültigen Parameterkombinationen erstellt:

real_clk steht für die effektive Taktrate in KHz, sollte im Bereich zwischen 3-15MHz sein.

dcm_clock_div Teiler der Uhr des FPGAs, wird für die Einstellung im Syntesewerkzeug Xilinx ISE benötigt.

oversampling Abtastung des `UART-RX` Signals, sollte nur im normalerweise nicht 16 sein

kep_clock_div Teiler der Uhr für den `UART` im *kep*$^{\varepsilon}$, sollte möglichst ganzzahlig sein, aber alle generierten Einträge sind gültig

Die beiden letzten Werte müssen in der Datei `data_types/data.strl` jeweils bei den Konstanten `oversampling` und `uart_clk_div` eingetragen werden. Man wählt also eine Kombination bei dem *real_ clk* zwischen 3000-15000 liegt, das *oversampling* möglichst 16 ist, und *kep_ clock_ div* möglichst ganzzahlig ist.

Zur Erzeugung von *VHDL*-Code kann das Makefile (`kep-e/Makefile`) mit zwei verschiedenen Optionen aufgerufen werden: hard oder hard_modular. Für jedes dieser beiden Optionen existiert im Esterel Studio-Projekt zum *kep*$^{\varepsilon}$ eine *Configuration*, die im Fall hard *monolithischen VHDL*-Code erzeugt und bei hard_modular *modularen VHDL*-Code erzeugt. Diese Konfigurationen geben auch den Grad der Optimierung an und sind auf *sweep* voreingestellt. Wenn andere Optimierungen eingestellt werden sollen, muss dies im Esterel Studio Projekt eingestellt werden (s. dazu Esterel Studio Handbuch [17]).

Monolithisch

`make hard.tiny.kep` erzeugt im Unterverzeichnis `hardkep/tiny` einen *kep*$^{\varepsilon}$ der Größe *tiny*. In diesesem Verz. befinden sich drei *VHDL*-Dateien:

- `esterel_numeric_std.vhd`
- `hardkep_data_pkg_template.vhd`
- `hardkep.vhd`

Analog gilt dies für die anderen Größen, bei selbst definierter Größe befindet sich der *VHDL*-Code im Unterverzeichnis `hardkep/custom`.

Modular

Die modulare Codegenerierung liefert bei höherer Optimierung effizienteren Code, terminiert jedoch nicht immer.

`make hard_modular.tiny.kep` erzeugt einen *kep*$^{\varepsilon}$ der Größe *tiny* im Unterverzeichnis `hardkep_modular/tiny`. In diesem verz. befinden sich mehrere *VHDL* Dateien.

A.1.2 *VHDL* Synthese

Projekt anlegen

Entsprechend des Ziel-FPGAs wird in Xilinx ISE [38] ein Projekt angelegt und die *VHDL* Dateien aus Abschnitt A.1.1 dem Projekt hinzugefügt. Für den *kep*$^{\varepsilon}$ wird ein *Schematic* Symbol erstellt.

Speicher erzeugen

Über den *Coregen Wizzard* werden zwei *Single-Port Block Memory Cores* [37] erzeugt. Für den Befehlsspeicher ist die Breite (*Width*) auf 40 zu setzen, da eine Instruktion 40bit lang ist. Die Tiefe (*Depth*) bestimmt sich aus der Größe des *kep*$^{\varepsilon}$: log_2(`INSTR`), wobei `INSTR` die Anzahl der Instruktionen aus der *kep*$^{\varepsilon}$-Konfiguration ist (s. Abschnitt A.1). Für den Ausführungsspeicher (*Tracebuffer*) wird als Breite die Tiefe des Befehlsspeichers angegeben. Die Tiefe bestimmt sich durch log_2(`TRACE`), wobei `TRACE` die Größe des Ausführungsspeichers aus der *kep*$^{\varepsilon}$-Konfiguration ist (s. Abschnitt A.1). Alle anderen Einstellungen werden so belassen.

Clockmanager

Über den *Coregen Wizzard* wird ein *Single DCM* erzeugt. Es müssen drei Einstellungen vorgenommen werden:

Input Clock Frequency wird auf die verwendete Uhr des FPGAs eingestellt.

CLK DV Ausgang auswählen

Divide by Value wurde mit dem Script calc_clk.pl aus Abschnitt A.1.1 bestimmt.

Alle anderen Einstellungen werden belassen. Anschließend wird für den Clockmanager ein *Schematic* Symbol erstellt.

Schematic

Dem Projekt wird ein neues, leeres *Schematic* hinzugefügt. In dem graphischen Editor werden alle generierten Symbole aus den vorherigen Schritten dem *Schematic* hinzugefügt. Jetzt müssen alle Pins nach Tabelle A.3 verbunden werden. *TRACE* und *ROM* sind die Namen der Speicher. Bei den Objekten *input* und *output* müssen für die Namen entsprechende Labels erstellt werden.

Objekt	Pin		Objekt	Pin
input	clk	→	dcm	CLKIN_IN
dcm	CLKDV_OUT	→	hardkep	clk
dcm	CLKDV_OUT	→	ROM	clk
dcm	CLKDV_OUT	→	TRACE	clk
input	rst	→	hardkep	rst
input	RX	→	hardkep	RX
ROM	dout	→	hardkep	instr_from_rom
TRACE	dout	→	hardkep	pc_val_from_buffer
hardkep	rom_addr_data	→	ROM	addr
hardkep	instr_to_rom	→	ROM	din
hardkep	wea_instr	→	ROM	we
hardkep	trace_buffer_addr	→	TRACE	addr
hardkep	pc_val_to_buffer	→	TRACE	din
hardkep	wea_trace	→	TRACE	we
hardkep	TX	→	output	TX
hardkep	no_rx_error	→	output	rx_error
hardkep	tx_busy	→	output	tx_busy
hardkep	no_tx_error	→	output	tx_error
hardkep	tick_warn	→	output	tick_warn

Tabelle A.3: Pinverbindungen für den *kep*$^{\varepsilon}$ und seine Speicher

User Constraint - File

Die *input* und *output* Labels des *Schematic* werden in eine `*.ucf` eingetragen. Unter `kep-e/hardkep/kepe_ucf.ucf` ist eine solche Datei bereits vorhanden, jedoch muss zumindest der Pin `clk, RX, TX` auf den richtigen Pin des FPGAs zugeordnet sein. Gleiches gilt für restlichen Ausgaben, jedoch können diese auch weggelassen werden. Fehlerhafte Zuordnungen können den *FPGA* beschädigen.

Es folgt die Synthese des Projekts die eine `*.bit`-Datei erstellt, die auf den *FPGA* geladen werden kann.

A.1.3 *softkep*$^{\varepsilon}$

Für den *softkep*$^{\varepsilon}$ gelten auch die Konfigurationen aus Abschnitt A.1. Zur Erzeugung von *C*-Code kann das Makefile (`kep-e/Makefile`) mit der Optionen soft aufgerufen werden. Es existiert im Esterel Studio-Projekt zum *kep*$^{\varepsilon}$ eine *Configuration* für dieses Ziel. Diese Konfiguration gibt den Grad der Optimierung an und ist auf *sweep* voreingestellt. Wenn andere Optimierungen eingestellt werden sollen, muss dies im Esterel Studio Projekt eingestellt werden (s. dazu Esterel Studio Handbuch [17]).

`make soft.tiny.kep` erzeugt im Unterverzeichnis `softkep/` eine ausführbare Datei `softkep.tiny`. Analog gilt dies für die anderen Größen, bei selbst

definierter Größe heißt die ausführbare Datei `softkep.custom`.

Die Erzeugung kann unter Linux als auch unter Cygwin mit dem Makefile durchgeführt werden.

Benutzung mit der `KepEvalBench`

Für die Benutzung des `KepEvalBench` mit dem *softkep$^{\varepsilon}$* werden zwei PCs benötigt, die über ein *Null-Modem-Kabel* miteinander verbunden sind. Alternativ kann man eine solche Nullmodemverbindung auch mit geeigneten Emulatoren herstellen.

B Protokoll des Testdrivers

Der Testdriver wird über ein einfaches Protokoll angesprochen. Es werden hier kurz alle Funktionen vorgestellt, die der Testdriver unterstützt. Für detailliertere Informationen kann entweder der Quellcode oder der Anhang B.2 der Dissertation von Li [26] studiert werden.

Verify Testet die Verbindung zum Testdriver und angeschlossenen PC.

```
–> "V"
<– "0123456789ABCDEFX" % '0' ist das erste Zeichen der Antwort.
```

On-board *kep*$^{\varepsilon}$ Konfiguration Gibt Informationen über die Konfiguration des kep^{ε} wieder. Relevante Felder sind:

- `KEP Type`
- `Signal number`
- `Thread number`
- `Delay counter width`
- `Tick length`
- `Instruction memory address width`.

```
–> "N"
<– "02001E1404040C040408041F407D00AX"
   % Codierung dieser Informationen
```

Programme in den Speicher laden Über `kasm2lst` compilierte Programme können in den Instruktions ROM geladen werden

```
–> "W"
–> "6000040000001E"    % '6' = Befehl wird das erste mal gesendet
–> "7000040000001E"    % '7 '= Befehl wird ein zweites mal gesendet
                       % Wenn die Uebertragungen nicht gleich sind wird
                       % <–"X" gesendet
–> "60001081800000"    % AWAIT R (0x081800000) an Adresse 0001
–> "70001081800000"
...
–> "60010080010000"    % HALT (0x080010000) an Adresse 0010
–> "70010080010000"


–> "60011000010002"    % GOTO A0 (0x000010002) an Adresse 0011
–> "70011000010002"
–> "X"                 % alle Instruktionen erfolgreich gespeichert
```

Neustarten des *kep*$^{\varepsilon}$ Startet den kep^{ε} neu, Testdriver muss dafür noch reagieren.

```
–> "R"          % Reset
<– "X"          % Aktion ist beendet
```

Setzen der Signalzustände Setzt die Eingangssignale des Prozessors. Da der *kep*$^{\varepsilon}$ nicht zwischen Ein- und Ausgabe unterscheidet, muss darauf geachtet werden, nur Eingaben zu setzen.

```
–> "I"          % Beginn des Setzens
–> "6"          % Codierte Information, keine direkte Adresse
–> "X"          % alle Zustaende sind abgefragt
```

Tick starten Startet eine neue Instanz und warte bis zum Ende der Instanz

```
–> "T"          % Startet die Instanz
<– "X"          % markiert das Ende der Instanz
```

Abfragen der Signalzustände Es werden alle Zustände (Ein- und Ausgänge sowie lokale) abgefragt. Die Zuordnung, welche Ein- oder Ausgänge bzw. lokale Signale sind, muss anschließend stattfinden.

```
–> "O"          % Start der Abfrage
<– "8"          % Codierte Information, keine direkte Adresse
<– "X"          % Ende der Uebertragung
```

Abfragen der aktuellen Länge der Instanz Gibt die Anzahl der verarbeitetet Instruktionen der letzten Instanz aus.

```
–> "L"          % Anfrage
<– "A000"       % Codierte Antwort
<– "X"          % Ende der Uebertragung
```

Auslesen des Tracebuffers Sendet den Inhalt des Ausführungsspeichers (*execution trace*)

```
–> "M"          % Anfrage
<– "38104810"   % Codierter Inhalt
<– "X"          % Ende der Uebertragung
```

Das Auslesen und Schreiben von *valued Signals* ist implementiert, wird jedoch nicht genutzt. Vor jedem Schreiben oder Lesen muss zunächst das entsprechende Signal ausgewählt werden.

C KASM Grammatik

Durch die formalisierte Grammatik wird KASM eindeutig und eine Abstraktionsschicht zwischen Esterel und dem *kep*$^{\varepsilon}$ geschaffen. Damit wird die Kommunikation zwischen der Entwicklung des Compilers und der Entwicklung des Prozessors verbessert. Die hier angegebene Grammatik ist die Definitionsdatei für den Parsergenerator `SableCC` [19].

C.0.4 kasm.grammar

```
Package kasm;

Helpers
  all    = [0x0 .. 0xfffff] ;
  digit  = ['0' .. '9'] ;
  lowercase= ['a' .. 'z'] ;
  uppercase= ['A' .. 'Z'] ;
  alpha  = [lowercase + uppercase] ;
  u_score = '_';
  dash   = '-';
  extalpha = [u_score + alpha] ;
  tab    = 9;
  lf     = 10;
  cr     = 13;
  eol    = cr lf | cr | lf ;
  not_cr_lf= [all - [cr + lf]] ;
  percent = '%' ;

  identifer= extalpha (extalpha | digit)* ;
  number = digit+ ;

  a = ['a' + 'A'];
  b = ['b' + 'B'];
  c = ['c' + 'C'];
  d = ['d' + 'D'];
  e = ['e' + 'E'];
  f = ['f' + 'F'];
  g = ['g' + 'G'];
  h = ['h' + 'H'];
  i = ['i' + 'I'];
  j = ['j' + 'J'];
  k = ['k' + 'K'];
  l = ['l' + 'L'];
  m = ['m' + 'M'];
  n = ['n' + 'N'];
  o = ['o' + 'O'];
  p = ['p' + 'P'];
  q = ['q' + 'Q'];
  r = ['r' + 'R'];
  s = ['s' + 'S'];
  t = ['t' + 'T'];
  u = ['u' + 'U'];
  v = ['v' + 'V'];
  w = ['w' + 'W'];
  x = ['x' + 'X'];
  y = ['y' + 'Y'];
  z = ['z' + 'Z'];

Tokens
  blank  = (' ' | tab | lf | cr)+ ;
  comment = percent not_cr_lf* eol ;

  colon  = ':' ;
  lparam = '(' ;
  rparam = ')' ;
  question = '?' ;
  pound  = '#' ;
  comma  = ',' ;


  ticklen = '_TICKLEN' ;
  //count  = '_COUNT' ;
  //tick   = t i c k ;
  nothing = n o t h i n g ;
  goto   = g o t o ;
  call   = c a l l ;
  return = r e t u r n ;

  present = p r e s e n t ;

  await  = a w a i t ;
  pause  = p a u s e ;
  awaiti = a w a i t i ;

  halt   = h a l t;

  cawaits = c a w a i t s ;
  cawait = c a w a i t ;
  cawaiti = c a w a i t i ;
  cawaite = c a w a i t e ;

  abort  = a b o r t ;
  aborti = a b o r t i ;
  wabort = w a b o r t ;
  waborti = w a b o r t i ;

  suspend = s u s p e n d ;
  suspendi = s u s p e n d i ;

  emit   = e m i t ;
  setv   = s e t v ;
  sustain = s u s t a i n ;

  par    = p a r ;
  pare   = p a r e ;
  prio   = p r i o ;

  join   = j o i n ;

  labort = l a b o r t ;
```

```
laborti = l a b o r t i ;
lwabort = l w a b o r t ;
lwaborti = l w a b o r t i ;

tabort = t a b o r t ;
taborti = t a b o r t i ;
twabort = t w a b o r t ;
twaborti = t w a b o r t i ;

jw     = j w ;
jc     = j c ;
jnc    = j n c ;

clrc   = c l r c ;
setc   = s e t c ;
sr     = s r ;
src    = s r c ;
sl     = s l ;
slc    = s l c ;
notr   = n o t r ;

load   = l o a d ;

add    = a d d ;
addc   = a d d c ;
sub    = s u b ;
subc   = s u b c ;
mul    = m u l ;

andr   = a n d r;
orr    = o r r ;
xorr   = x o r r ;

cmp    = c m p ;
cmps   = c m p s;

def32  = d e f '32' ;

exit   = e x i t;

input  = i n p u t;
output = o u t p u t;
inputv = i n p u t v ;
outputv = o u t p u t v;
signal = s i g n a l;
signalv = s i g n a l v;
var    = v a r ;
pre    = p r e ;

number = number;
identifer= identifer;

z      = z ;
l      = l ;
g      = g ;
ge     = g e ;
le     = l e ;

ee     = e e ;
ne     = n e ;



Ignored Tokens
 blank ,
 comment ;

Productions
 program = io_expr* T.emit T.ticklen comma T.pound data kst_list*
    ;
 kst_list = {kasm_stat} kasm_st
    |     {label}   addr    colon
    ;
 io_expr = {input}  T.input sig_list
    |     {output}  T.output sig_list
    |     {inputv}  T.inputv sig_list
    |     {outputv} T.outputv sig_list
    |     {signal}  T.signal sig_list
    |     {signalv} T.signalv sig_list
    |     {var}     T.var   var_list
    ;
 sig_list = {single} signal_id
    |     {list}    sig_list comma signal_id
    ;

 var_list = {single} variable
    |     {list}    var_list comma variable
    ;

 kasm_st = {nothing} T.nothing
    |     {goto}    T.goto  addr
    |     {call}    T.call  addr
    |     {return}  T.return

    |     {present} T.present P.signal comma addr

    |     {await}   T.await P.signal
    |     {pause}   T.pause  //[f]:prio_val comma [s]:prio_val
    |     {awaiti}  T.awaiti P.signal
    |     {halt}    T.halt

    |     {cawaits} T.cawaits
    |     {cawait}  T.cawait P.signal comma addr
    |     {cawaiti} T.cawaiti P.signal comma addr
    |     {cawaite} T.cawaite addr

    |     {abort}   T.abort P.signal [f]:comma addr
    |     {aborti}  T.aborti P.signal [f]:comma addr
    |     {wabort}  T.wabort P.signal [f]:comma addr
    |     {waborti} T.waborti P.signal [f]:comma addr

    |     {suspend} T.suspend P.signal [f]:comma addr
```

```
    |   {suspendi} T.suspendi P.signal [f]:comma addr

    |   {signalin_pure} T.signal signal_id
    |   {signalin_val} T.signalv signal_id
    |   {emit}    T.emit  sig_expr
    |   {setv}    T.setv  sig_expr
    |   {sustain} T.sustain sig_expr
    |   {par}     T.par   prio_val [f]:comma addr [s]:comma thread_id
    |   {pare}    T.pare  addr   comma  prio_val
    |   {prio_w_th} T.prio prio_val comma thread_id
    |   {prio_wo_th} T.prio prio_val
    |   {join}    T.join  prio_val

    |   {labort}  T.labort P.signal [f]:comma addr
    |   {laborti} T.laborti P.signal [f]:comma addr
    |   {lwabort} T.lwabort P.signal [f]:comma addr
    |   {lwaborti} T.lwaborti P.signal [f]:comma addr

    |   {tabort}  T.tabort P.signal [f]:comma addr
    |   {taborti} T.taborti P.signal [f]:comma addr
    |   {twabort} T.twabort P.signal [f]:comma addr
    |   {twaborti} T.twaborti P.signal [f]:comma addr

    |   {jw}      T.jw    cond_expr
    |   {jc}      T.jc    addr
    |   {jnc}     T.jnc   addr
    |   {clrc}    T.clrc
    |   {setc}    T.setc
    |   {sr}      T.sr    reg
    |   {src}     T.src   reg
    |   {sl}      T.sl    reg
    |   {slc}     T.slc   reg
    |   {notr}    T.notr  reg
    |   {load}    T.load  reg_expr
    |   {add}     T.add   reg_expr
    |   {addc}    T.addc  reg_expr
    |   {sub}     T.sub   reg_expr
    |   {subc}    T.subc  reg_expr
    |   {mul}     T.mul   reg_expr
    |   {andr}    T.andr  reg_expr
    |   {orr}     T.orr   reg_expr
    |   {xorr}    T.xorr  reg_expr
    |   {cmp}     T.cmp   reg_expr
    |   {cmps}    T.cmps  reg_expr
    |   {def32}   T.def32 T.pound data
    |   {exit}    T.exit  [from]:addr comma  [to]:addr
    ;

sig_expr = {pure}  signal_id
    |   {data}    signal_id comma T.pound data
    |   {reg}     signal_id comma reg
    ;

cond_expr= {z}     T.z     comma  addr
    |   {l}       T.l     comma  addr
    |   {g}       T.g     comma  addr
    |   {ge}      T.ge    comma  addr
    |   {le}      T.le    comma  addr
    |   {ee}      T.ee    comma  addr
    |   {ne}      T.ne    comma  addr
    ;

signal = {current} signal_id
    |   {pre}     T.pre   T.lparam signal_id T.rparam
    ;

reg_expr = {data}  reg       comma  T.pound data
    |   {reg}     [target]:reg comma [source]:reg
    |   {sig_val} reg       comma  signal_val
    ;

signal_val= {current} question signal_id
    |   {pre}     T.pre     lparam question signal_id rparam
    ;

addr    = identifer ;
signal_id = identifer ;
reg     = identifer ;
variable = identifer ;
prio_val = number;
thread_id = number;
data    = number;
watcher = number;
```

D Esterel-Quellcode

D.1 *kep*$^{\varepsilon}$

D.1.1 Verarbeitungszyklus

D.1.2 kepe.strl

```
module kep_e:

  extends instr_data;
  extends signal_data;

  // Watcher
  extends watcher_data;

  // Threads
  extends thread_data;
  extends await_data;

  output   Tick,tick_warn;
  input    tick_start;

  // signal status
  input    env_signals      : temp value signal_status_type;
  output   pre_signals      : temp value signal_status_type;

  // instr. rom interface
  output   rom_addr         : temp    rom_addr_type;
  input    instr_from_rom   : temp value instr_type;


  // signal ram interface
  // output  signal_ram_addr  : temp value signal_addr_type ;
  // output  signal_val_to_ram : temp value signal_value_type;
  // input   signal_val_from_ram : temp value signal_value_type;
  // output  wea_signal;

  output instr_count        :   value unsigned<[trace_buffer_addr_width]> init 0;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

  signal


    // signal status
    cur_signals           :     value signal_status_type init '0,


    // op_code interface
    extends op_code_intf,


    // Threads
    extends thread_intf,

    // tick_manager
    tick_finished,
    tick_ready,
    thread_ready,
    tick_real_ready,
    dec_tick_len,
    set_tick_len          :         instr_data_type init 1,

    // program_counter
    load_pc               : temp    program_counter_type init 0,
    load_pc_ctsw          : temp    program_counter_type init 0,
    pc                    :    value program_counter_type init 0,
    enable_pc,

    // watcher
    watcher[max_watcher]  :    value watcher_type  init '0,
    tws[max_watcher]      : temp    program_counter_type,
    tws_type[max_watcher] : temp value preemption_type,
    watcher_abort_id      : temp value watcher_id_type,
    watcher_abort,
    suspend_active        : temp   watcher_id_type,
    watcher_id            : temp    watcher_id_type ,

    // threads
    threads[max_threads]  : value   thread_type init '0,
    cur_thread_id         : value   unsigned<[thread_width]> init 0,
   // pre_thread_id         : value   unsigned<[thread_width]> init 0,
    thread_init,

    // await cell
    await_reg[max_threads] : value   await_type init '0,

    // _count signal
    load_count            : value   delay_counter_type init u2bin(1,
   delay_counter_width),
    reset_load_count

  in

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

    // set up main thread
    emit {
       ?threads[?cur_thread_id].running <=true,
       ?threads[?cur_thread_id].active <=true,
       ?threads[?cur_thread_id].end_addr <= {rom_addr_width{'1}}

    };
    //emit env_signals(reset_signal_const);
    //emit pre_signals(reset_signal_const);

    run watcher_controller_fast
    ||
    // Instantiating watcher via script to observe watcher closer in the simulator
```

```
//<#watcher-script>
/*   run watcher_0 / watcher_cell[constant 0/my_id; tws[0]/tw; tws_type[0]/
tw_type]
   ||
   run watcher_1 / watcher_cell[constant 1/my_id; tws[1]/tw; tws_type[1]/tw_type
]
   ||
   run watcher_2 / watcher_cell[constant 2/my_id; tws[2]/tw; tws_type[2]/tw_type
]
   ||
   run watcher_3 / watcher_cell[constant 3/my_id; tws[3]/tw; tws_type[3]/tw_type
]
 */// ||
//<#/watcher-script>

// script-free version for instantiating watcher modules
for i < max_watcher dopar
   run watcher_cell [constant i/my_id; tws[i]/tw; tws_type[i]/tw_type]
end for
||

// instantiating module for creating Threads
run thread_create
||
// instantiating module for watching Threads
run watch_tick_ready
||
// instantiating module for killing Threads
run finish_thread
||
//instantiating scheduler
run ctsw_fast
/*||
// not needed anymore
run thread_id_relay
*/||
// re-aktivate threads
run run_threads
||
// instantiating module for killing Threads
run term_threads
||
// instantiating module for controlling env. io
run io_controller
||
// instantiating tick_manager
run tick_manager
||
// instantiating program_counter
run program_counter;
||
pause;
// main execution loop
loop
 await Tick; // wait for begin of a macro-tick, if not Tick

weak abort


    [run fetch; // module for fetching a instruction from the ROM
    ||
     // update the instruction counter (just for debugging purpose)
    pause;
    emit instr_count(sat<[trace_buffer_addr_width]>(pre(?instr_count)+1));
    ];


    run decode; // decode the fetched instruction
    run execute; // execute the decoded instruction

    emit dec_tick_len; // instruction is executed => decrement tick_len

 when immediate (watcher_abort or suspend_active) do
 if suspend_active and not watcher_abort then
       pause; // ensure equiditant (CPI) time
       pause;
       pause;
       emit tick_ready;
       emit dec_tick_len;
   else
       pause; // ensure equiditant (CPI) time
       pause;
       pause;
       emit dec_tick_len;
 end if;

  /*  pause; // wait one tick to ensure equidistant time per instruction
    emit {
       tick_ready <=   pre(suspend_active) and not pre(watcher_abort),
       dec_tick_len <= not tick_ready //(pre(suspend_active) and not pre(
watcher_abort))
       }
   */
 end abort;



   if tick_real_ready then
      // wait for remaining ticks of tick_len
      await immediate tick_finished
   end if;

  end loop
 end signal

end module
```

D.1.3 fetch.strl

```
module fetch:

% fetch a instr. from the instr.rom

   extends instr_data;
   extends thread_data;

   input pc                   : temp value program_counter_type;
   output rom_addr            : temp    rom_addr_type;

   input watcher_abort;
   // threads
   output threads[max_threads] : temp value thread_type;
   input cur_thread_id        : temp value unsigned<[thread_width]>;

%-----------------------------------------------------------------------------

   // query instr_rom at addr of cur. PC val
   // this also sets the trace buffer
   emit ?rom_addr <= u2bin(?pc);

   // save the PC. to the cur thread PC
   emit ?threads[?cur_thread_id].cur_addr <= u2bin(?pc) if not watcher_abort;

   pause; // wait for the answer of instr_rom and count the instr.
   pause; // wait another cycle, because blockRam needs a least one cylce


end module
```

D.1.4 decoder.strl

```
module decode :

% decodes the fetched instr. and emits a signal to excecutes the fitting module

   extends instr_data;

   extends op_code_intf;
   extends thread_intf;

   // fetched instruction
   input    instr_from_rom    : temp value instr_type;

%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------

   emit {
      ?EMIT       <= lcat(?instr_from_rom[signal_start..signal_end],
                          ?instr_from_rom[sig_data_start..sig_data_end]) if ?
      instr_from_rom.id = 'b01000000,
      ?SIGNALIN   <=    ?instr_from_rom[signal_start..signal_end] if ?instr_from_rom.
      id = 'b00010000,
      ?SUSTAIN    <= lcat(?instr_from_rom[signal_start..signal_end],
                          ?instr_from_rom[sig_data_start..sig_data_end]) if ?
      instr_from_rom.id = 'b01001000,
       NOTHING    <=                                          ?instr_from_rom.id = '
      b00000000,
       PAUSE      <=                                          ?instr_from_rom.id = '
      b00001010,
      ?GOTO       <=    ?instr_from_rom[addr_start..addr_end] if ?instr_from_rom.id =
      'b00000001,
/*EXIT*/?GOTO     <=    ?instr_from_rom[addr_start..addr_end] if ?instr_from_rom.id =
      'b00000101,
      ?PRESENT    <= lcat(?instr_from_rom[addr_start..addr_end],
                          ?instr_from_rom[addr_sig_start..addr_sig_end]) if ?
      instr_from_rom.id = 'b00000110,
      ?ABORT      <= lcat(?instr_from_rom[addr_start..addr_end],
                          ?instr_from_rom[addr_sig_start..addr_sig_end],
                          ?instr_from_rom[addr_sig_w_start..addr_sig_w_end],
                          ?instr_from_rom.id[0..3])           if ?instr_from_rom.
      first4id = 'b1000,
       HALT       <=                                          ?instr_from_rom.id = '
      b00001011,
      ?LOAD_REG_DATA <= lcat(?instr_from_rom[reg_start..reg_end],
                             ?instr_from_rom[reg_data_start..reg_data_end]) if ?
      instr_from_rom.id = 'b11010000,
      ?AWAIT      <= lcat(?instr_from_rom.id[0..0],
                          ?instr_from_rom[signal_start..signal_end]) if ?instr_from_rom.
      id[1..7] = 'b0000100,
      // Threads
      ?PAR        <= lcat(?instr_from_rom[addr_start..addr_end],
                          ?instr_from_rom[addr_th_start..addr_th_end],
                          ?instr_from_rom[addr_th_pr_start..addr_th_pr_end]) if ?
      instr_from_rom.id = 'b01010000,
      ?PARE       <= lcat(?instr_from_rom[addr_start..addr_end],
                          ?instr_from_rom[addr_prio_start..addr_prio_end]) if ?
      instr_from_rom.id = 'b01010001,
      ?PRIO       <= lcat(?instr_from_rom[th_start..th_end],
                          ?instr_from_rom[th_pr_start..th_pr_end]) if ?instr_from_rom.id
      = 'b01010010,
      ?JOIN       <= ?instr_from_rom.para[24..23+thread_prio_width] if ?instr_from_rom
      .id = 'b01010011,
       no_op      <= not( EMIT
                          or SIGNALIN
                          or SUSTAIN
                          or NOTHING
                          or PAUSE
                          or GOTO
                          or PRESENT
                          or HALT
                          or ABORT
                          or LOAD_REG_DATA
                          or AWAIT
                          or JOIN
                         )
   }

end module
```

D.1.5 execute.strl

```
module execute :

% executes fitting run module with the decoded instr.

   extends instr_data;
   extends signal_data;
   extends data_data;
   extends thread_data;
   extends prio_data;
   extends watcher_data;
   extends await_data;

   // signal status
   inputoutput cur_signals    : temp value signal_status_type;
   inputoutput pre_signals    : temp value signal_status_type;

   extends mirror op_code_intf ;

   // Threads
   input    JOIN              : temp    thread_prio_type;
   input    PRIO              : temp    prio_type;
   input relation execute_thread_rel: JOIN # PRIO;

   inputoutput threads[max_threads] : temp value thread_type;
   inputoutput await_reg[max_threads] : temp value await_type;
   input    cur_thread_id     : temp value unsigned<[thread_width]>;

   // signal RAM
   // output  signal_val_to_ram : temp value signal_value_type;
   // output  signal_ram_addr   : temp value signal_addr_type;
   // input   signal_val_from_ram : temp value signal_value_type;
   // output  wea_signal;

   // tickmanager
   input    Tick;
   output   tick_ready,thread_ready;
   output   load_pc           : temp program_counter_type;
   output   set_tick_len      : temp instr_data_type;

   // watcher
   output   watcher[max_watcher] : temp value watcher_type;
   output   tws[max_watcher] : temp    program_counter_type;
   output   tws_type[max_watcher] : temp value preemption_type;
   output   watcher_id       : temp    watcher_id_type;
   input    watcher_abort;

   // _count signal
   inputoutput load_count    : temp value delay_counter_type;
   output reset_load_count;

   // program counter
   input    pc               : temp value program_counter_type;
   output   enable_pc;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

       run execute_emit    [ EMIT      / instr_in ]
       ||
       run execute_signalin [ SIGNALIN / instr_in ]
       ||
       run execute_sustain [ SUSTAIN   / instr_in ]
       ||
       run execute_nothing [ NOTHING   / enable ]
       ||
       run execute_halt    [ HALT      / enable ]
       ||
       run execute_present [ PRESENT   / instr_in ]
       ||
       run execute_abort   [ ABORT     / instr_in ]
       ||
       run execute_goto    [ GOTO      / instr_in ]
       ||
       run execute_no_op_pc [ no_op    / enable ]
       ||
       run execute_join    [ JOIN      / instr_in ]
       ||
       run execute_prio    [ PRIO      / instr_in ]
       ||
       run await_cell
       ||
       run execute_load_reg_data [ LOAD_REG_DATA / instr_in ]

end module
```

D.1.6 programcounter.strl

```
module program_counter:

% sets the program_counter:
% 1. to the next instruction (+1)
% 2. to a specific address (load_pc)
% 3. to the next thread

   extends instr_data;
   extends thread_data;
   extends watcher_data;

   input enable_pc;
   input load_pc        : temp    program_counter_type;
   input load_pc_ctsw   : temp    program_counter_type;
   input threads[max_threads] : temp value thread_type;
   input cur_thread_id  : temp value unsigned<[thread_width]>;
   output pc            :      reg program_counter_type init 0;

   %-------------------------------------------------------------------------
   %-------------------------------------------------------------------------

   sustain {
      if enable_pc then

         next ?pc <= ?load_pc_ctsw      if load_pc_ctsw,
         next ?pc <= ?load_pc           if load_pc and not load_pc_ctsw,
         next ?pc <= sat<[rom_addr_width]>(?pc+1) if not load_pc and
                                          not load_pc_ctsw and
                                          not ?threads[?cur_thread_id].delayed

      end if
      }

end module
```

D.1.7 Der Reaktive Kern

D.1.8 iocontroller.strl

```
module io_controller:

% 1. copies environment to current signal status
% 2. copies current to previous signal status

   extends signal_data;

   input        tick_finished,
                tick_start;
   input relation io_contr_rel_1: tick_finished # tick_start;


   output instr_count        : temp value unsigned<[trace_buffer_addr_width]>;

   inputoutput cur_signals   : temp value signal_status_type; // signal status of the
      CURRENT Tick
   output    pre_signals     : temp value signal_status_type; // signal status of the
      PREVIOUS Tick
   input     env_signals     : temp value signal_status_type; // signal status of the
      NEXT Tick (set by the environment)

%--------------------------------------------------------------------------


%--------------------------------------------------------------------------



   // copy environment to current status
   loop
      await tick_start;
      emit {
         cur_signals(?env_signals),
         instr_count(0)
      };
   end loop
   ||
   // copy current to previous status
   loop
      await tick_finished;
      emit pre_signals(?cur_signals);
      //pause;
      //emit cur_signals(reset_signal_const);
   end loop

end module
```

D.1.9 emit.strl

```
module execute_emit :

% emits a signal encoded in instr_in:
% EMIT Signal #Data
%
% 01000000 SSSSSSSSP NNNNNNNNNNNNNNN (34)
% module takes no time for computing

   extends instr_data;
   extends signal_data;
   extends sig_data_data;


   // current instruction
   input instr_in     : temp    sig_data_type;

   // signal ram
   // output wea_signal;
   // output signal_val_to_ram : value temp signal_value_type;
   // output signal_ram_addr : value temp signal_addr_type;

   // signal status
   output cur_signals : value temp signal_status_type;
   // set ticklen
   output set_tick_len : temp   instr_data_type;
   // program counter
   output enable_pc;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   if instr_in then

      var sig_id : temp unsigned<[signal_addr_width]> // just for indexing
      in
         sig_id:=bin2u(?instr_in.signal_id);
         emit cur_signals[sig_id](true);
         // emit signal_ram_addr(?instr_in.signal_id);
         // emit signal_val_to_ram(?instr_in.carried_data);
         // emit wea_signal;


         // if signal is _ticklen then set ticklen
         if sig_id=ticklen_signal_addr_const then
            emit set_tick_len(bin2u(?instr_in.carried_data))
         end if;

      end var;

      emit enable_pc; // update program_counter++
      await 2 tick; // delay 2 tick to have equidistant time per instruction

   end if

end module
```

D.1.10 sustain.strl

```
module execute_sustain :


% like emit, but marks the thread as ready for this tick and has to wait for the
    watcher
% SUSTAIN Signal #Data
% 01001000 SSSSSSSSP NNNNNNNNNNNNNNN (34)
% module takes 2 tick for computing and marks the thread as ready for this tick

  extends instr_data;
  extends signal_data;
  extends sig_data_data;


  input instr_in     :      sig_data_type;

  /*// signal ram
    output wea_signal;
    output signal_val_to_ram : value temp signal_value_type;
    output signal_ram_addr : value temp signal_addr_type;
  */
  // signal status
  output cur_signals : value temp signal_status_type;


  output tick_ready;

%-----------------------------------------------------------------------------
%-----------------------------------------------------------------------------

  if instr_in then

    await 2 tick; // wait for watcher computation, if a watcher is fired, the module
     will be aborted

    var sig_id : temp unsigned<[signal_addr_width]>
    in
      sig_id:=bin2u(?instr_in.signal_id);
      emit cur_signals[sig_id](true);
      /* emit signal_ram_addr(?instr_in.signal_id);
        emit signal_val_to_ram(?instr_in.carried_data);
        emit wea_signal;
       */
    end var;

    // do not program_counter
    emit tick_ready;


  end if

end module
```

D.1.11 signalin.strl

```
module execute_signalin :

% reset local signal
% SIGNAL Signal
%
% 00010000 SSSSSSSSP (18)
% module takes no time for computing

   extends signal_data;


   // current instruction
   input instr_in     : temp    sig_type;

   // signal status
   output cur_signals : value temp signal_status_type;
   output pre_signals : value temp signal_status_type;

   // program counter
   output enable_pc;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   if instr_in then

      var sig_id : temp unsigned<[signal_addr_width]>
      in

         sig_id:=bin2u(?instr_in.signal_id);
         emit cur_signals[sig_id](false);
         emit pre_signals[sig_id](false);

      end var;
        emit enable_pc; // update program_counter++
        await 2 tick; // delay 2 tick to have equidistant time per instruction


   end if

end module
```

D.1.12 present.strl

```
module execute_present :

% test status of a signal, if negative branch(load_pc)
% PRESENT elseLabel, Signal
%
% 00000110 AAAAAAAAAAAAAAAA SSSSSSSSP (34)
% module takes no time for computing


   extends instr_data;
   extends signal_data;
   extends sig_addr_data;

   // signal status
   input cur_signals : temp value signal_status_type;
   input pre_signals : temp value signal_status_type;
   // set else Addr
   output load_pc : temp program_counter_type;
   // current instruction
   input instr_in : temp sig_addr_type;
   // program counter
   output enable_pc;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------
   if instr_in then

      var sig_id : temp unsigned<max_signals>
      in
         sig_id:=bin2u(?instr_in.signal_id);
         // test if the signal is pre
         if ?instr_in.pre_op then
               // test the previous signal status
               // if test negative
               // set pc to else addr
               emit load_pc(bin2u(?instr_in.addr)) if not ?pre_signals[sig_id]
         else
               // test the current signal status
               // if test negative
               // set pc to else addr
               emit load_pc(bin2u(?instr_in.addr)) if not ?cur_signals[sig_id]

         end if
      end var;
      emit enable_pc; // update program_counter
      await 2 tick; // delay 2 tick to have equidistant time per instruction

   end if

end module
```

D.1.13 goto.strl

```
module execute_goto :

% set the pc to the address encoded in instr_in
% GOTO Label
%
% 00000001 AAAAAAAAAAAAAAA (24)
% module takes no time for computing

  extends instr_data;

  // current instruction
  input instr_in : temp rom_addr_type;
  // set goto-addr
  output load_pc : temp program_counter_type;
  // program counter
  output enable_pc;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

  if instr_in then
       // set goto-addr
       emit load_pc(bin2u(?instr_in));

       emit enable_pc; // update program_counter
       await 2 tick; // delay 2 tick to have equidistant time per instruction

  end if

end module
```

D.1.14 noop.strl

```
module execute_no_op_pc :

% NO OPERATION, just set the pc to the next instruction
%
% has no opcode
% module takes no time for computing

   input enable ;
   // program counter
   output enable_pc;

%--------------------------------------------------------------------------

   if enable then

      emit enable_pc; // update program_counter++
      await 2 tick; // delay 2 tick to have equidistant time per instruction

   end if

end module
```

D.1.15 nothing.strl

```
module execute_nothing :

% NOTHING
%
% 00000000 (8)
% module takes no time for computing

   input enable ;
   // program counter
   output enable_pc;

%--------------------------------------------------------------------------

   if enable then

      emit enable_pc; // update program_counter
      await 2 tick; // delay 2 tick to have equidistant time per instruction

   end if

end module
```

D.1.16 load.strl

```
module execute_load_reg_data :

% Load data into a register:a the moment only used for reading the load_count value
% LOAD Register, #Data
% 11010000 RRRRRRRRR NNNNNNNNNNNNNNN (34)
% module takes no time for computing


   extends instr_data;
   extends register_data;
   extends reg_data_data;

   input instr_in : temp reg_data_type;

   // _count signal
   output load_count : temp value delay_counter_type;

   // program counter
   output enable_pc;

%-------------------------------------------------------------------------------

   if instr_in then

      var reg_id :temp unsigned<[register_addr_width]>
      in
         reg_id:=bin2u(?instr_in.target);
         // if register is _count then set count
         emit load_count(?instr_in.carried_data[0..delay_counter_width-1]) if reg_id=
      count_id_const

      end var;


      emit enable_pc; // update program_counter
      await 2 tick; // delay 2 tick to have equidistant time per instruction

   end if

end module
```

D.1.17 await.strl

```
// AWAIT Signal
//
// 00001000 SSSSSSSSP (18)
// tick: 0
module execute_await :

   extends instr_data;
   extends await_data;
   extends thread_data;

   inputoutput await_reg[max_threads] : value await_type;
   input  cur_thread_id   : value unsigned<[thread_width]>;
   // current instruction
   input instr_in     : temp    sig_await_type;
  // indicate Tick is over
   output tick_ready;

   //output counter : temp value instr_data_type;

   // _count signal
   input load_count : reg value delay_counter_type;
   // program counter
  // output enable_pc;

   if instr_in then

     if not pre(?await_reg[?cur_thread_id].running) then
        // config await reg
        emit {

           ?await_reg[?cur_thread_id].sig_await <= ?instr_in,
           ?await_reg[?cur_thread_id].count <= ?load_count,
           ?await_reg[?cur_thread_id].running <= true
           };
      end if;
     // emit counter(?load_count);
     // do not update pc
     // emit enable_pc; // update program_counter
    // await 2 tick; // delay 2 tick to have equidistant time per instruction
     pause;
      emit tick_ready;
     //await await_done;

   end if

end module
```

D.1.18 halt.strl

```
module execute_halt :

% halts the exceution: do nothing
% HALT
%
% 00001011 (8)
% module takes no time for computing and marks the thread as ready for this tick

   input enable ;
   // indicate Tick is over
   output tick_ready;

%--------------------------------------------------------------------------
%--------------------------------------------------------------------------

   if enable then


      await 2 tick; // delay 2 tick to have equidistant time per instruction

      // mark tick as over
      emit tick_ready;
      // do not update pc


   end if

end module
```

D.1.19 abort.strl

```
module execute_abort :

% configures watcher, handles all types of preemption
%
% ABORT endLabel, Signal, Watcher
% 10000000 AAAAAAAAAAAAAAAA SSSSSSSSSP WWWWWW (40)
%
% ABORTI endLabel, Signal, Watcher
% 10000001 AAAAAAAAAAAAAAAA SSSSSSSSSP WWWWWW (40)
%
% WABORT endLabel, Signal, Watcher
% 10000010 AAAAAAAAAAAAAAAA SSSSSSSSSP WWWWWW (40)
%
% WABORTI endLabel, Signal, Watcher
% 10000011 AAAAAAAAAAAAAAAA SSSSSSSSSP WWWWWW (40)
%
% SUSPEND endLabel, Signal, Watcher
% 10000100 AAAAAAAAAAAAAAAA SSSSSSSSSP WWWWWW (40)
%
% SUSPENDI endLabel, Signal, Watcher
% 10000101 AAAAAAAAAAAAAAAA SSSSSSSSSP WWWWWW (40)

% module takes no time for computing

%------------------------------------------------------------------------
%------------------------------------------------------------------------

   extends instr_data;
   extends signal_data;
   extends sig_addr_watcher_data;
   extends watcher_data;

   // current instruction
   input    instr_in         : temp    sig_addr_watcher_type;

   // reset _count signal after using it for watcher init.
   output   reset_load_count;

   // signals
   input    cur_signals      : temp value signal_status_type;
   input    pre_signals      : temp value signal_status_type;

   // watcher
   output   watcher_id       : temp    watcher_id_type ;
   output   watcher[max_watcher] : temp value watcher_type;
   output   tws[max_watcher] : temp    program_counter_type ;
   output   tws_type[max_watcher] : temp value preemption_type ;

   // thread
   input    cur_thread_id    : temp value unsigned<[thread_width]>;

   // program counter
   output   enable_pc;
   input    pc               : temp value program_counter_type ;


if instr_in then

      var trig  : temp sig_type ,
          id    : temp watcher_id_type,
          s_index : temp unsigned<max_signals>

      in
          trig:=?instr_in.signal_code;
          id := bin2u(?instr_in.watcher_id);
          s_index:=bin2u(trig.signal_id);

          // write watcher configuration to the watcher array
          emit {
             ?watcher[id].start_addr <= u2bin(?pc),
             ?watcher[id].end_addr <= ?instr_in.addr,
             ?watcher[id].isPre <= trig.pre_op,
             ?watcher[id].trigger <= trig.signal_id,
             ?watcher[id].preemption <= STRONG if ?instr_in.typ='b000,
             ?watcher[id].preemption <= SUSPEND if ?instr_in.typ='b010 and not ?
   instr_in.immed,
             ?watcher[id].preemption <= I_SUSPEND if ?instr_in.typ='b010 and ?
   instr_in.immed,
             ?watcher[id].preemption <= WEAK if ?instr_in.typ='b001 and not ?
   instr_in.immed,
             ?watcher[id].preemption <= I_WEAK if ?instr_in.typ='b001 and ?instr_in.
   immed,
             ?watcher[id].thread_id <= u2bin(?cur_thread_id)
            // ?watcher[id].fired <= false
          };



       if (?instr_in.immed and ( ?instr_in.typ=STRONG)) //preemption is IMMEDIATE
   and STRONG
                       and ((( trig.pre_op and ?pre_signals[s_index]) //and trigger
    is present
                       or ((not trig.pre_op) and ?cur_signals[s_index])))
       then
           emit tws[id](bin2u(?instr_in.addr)); // do not config watcher => fire
   watcher
           emit tws_type[id](STRONG)

       else
           emit watcher_id(id); // configure watcher
           emit enable_pc; // update program_counter++
       end if
```

```
    end if
end module

end var;
await 2 tick; // delay 2 tick to have equidistant time per instruction
emit reset_load_count;
```

D.1.20 threadcontroller.strl

```
module run_threads:

% re-activates all still running threads at beginning of the new Tick

   extends thread_data;


   input tick_finished;

   inputoutput threads[max_threads] : value thread_type;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   every tick_finished do
      emit {
         for i < max_threads dopar
            ?threads[i].active <= pre(?threads[i].running)
         end for
         };
   end every

end module

module term_threads:

% terminate current running thread, if the pc is outside the scope

   extends thread_data;
   extends instr_data;
   input tick_start;

   input cur_thread_id : value unsigned<[thread_width]>;
   inputoutput threads[max_threads] : value thread_type;
   input pc :value program_counter_type ;
   input thread_init;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   await tick_start;
   suspend
      sustain {
         ?threads[?cur_thread_id].running <= false if
               bin2u(pre(?threads[?cur_thread_id].end_addr))<=?pc,
         ?threads[?cur_thread_id].active <= false if
               bin2u(pre(?threads[?cur_thread_id].end_addr))<=?pc
         };
   when thread_init;

end module
```

```
module ctsw:

% performs the ctsw analysis
% this the the slow version

   extends thread_data;
   extends instr_data;
   extends watcher_data;

   input pre_thread_id : temp value unsigned<[thread_width]>;
   output cur_thread_id : value reg unsigned<[thread_width]> init 0;
   input threads[max_threads] : temp value thread_type;

   input thread_init;
   output enable_pc ;
   output load_pc_ctsw : temp program_counter_type;
   input watcher_abort;


%-------------------------------------------------------------------------
%-------------------------------------------------------------------------


   suspend % when thread_init
      loop
         var max_prio : temp unsigned<[thread_prio_width]> := 0
         in
            var id : temp unsigned<[thread_width]> := ?pre_thread_id
            in
               //<thread-script>
               if ?threads[0].active and bin2u(?threads[0].prio)>=max_prio then
                     id:=0;
                     max_prio:=bin2u(?threads[0].prio)
               end if;
               if ?threads[1].active and bin2u(?threads[1].prio)>=max_prio then
                     id:=1;
                     max_prio:=bin2u(?threads[1].prio)
               end if;
               if ?threads[2].active and bin2u(?threads[2].prio)>=max_prio then
                     id:=2;
                     max_prio:=bin2u(?threads[2].prio)
               end if;
               if ?threads[3].active and bin2u(?threads[3].prio)>=max_prio then
                     id:=3;
                     max_prio:=bin2u(?threads[3].prio)
               end if;
               if ?threads[4].active and bin2u(?threads[4].prio)>=max_prio then
                     id:=4;
                     max_prio:=bin2u(?threads[4].prio)
               end if;
               if ?threads[5].active and bin2u(?threads[5].prio)>=max_prio then
```

```
                    id:=5;
                    max_prio:=bin2u(?threads[5].prio)
                end if;
                if ?threads[6].active and bin2u(?threads[6].prio)>=max_prio then
                    id:=6;
                    max_prio:=bin2u(?threads[6].prio)
                end if;
                if ?threads[7].active and bin2u(?threads[7].prio)>=max_prio then
                    id:=7;
                    max_prio:=bin2u(?threads[7].prio)
                end if;
                //</thread-script>
                % if the thread_id is the same as the running thread_id: do nothing
                if not ?pre_thread_id=id or watcher_abort then
                   emit {
                        next ?cur_thread_id <= id, % set the new thread
                        ?load_pc_ctsw       <= bin2u(?threads[id].cur_addr),
                        enable_pc %load the pc with the new addr
                        }
                   end if

            end var ;
         end var ;
         pause;
      end loop
   when thread_init
end module

module finish_thread:

% change status of the current thread, if it is ready in this tick
% also switch the delay state

   extends thread_data;

   input    cur_thread_id   : temp value unsigned<[thread_width]> ;
   inputoutput threads[max_threads] : value thread_type;

   input tick_ready, watcher_abort;
   output thread_ready;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   every tick_ready do
      emit {
         thread_ready,
         ?threads[?cur_thread_id].active <= false if not pre(?threads[?cur_thread_id].
      delayed) and not watcher_abort ,
         ?threads[?cur_thread_id].delayed <= not pre(?threads[?cur_thread_id].delayed)
       if not watcher_abort
      };
   end every

end module
```

```
module ctsw_fast:

% performs the ctsw analysis
% this the the slow version

   extends thread_data;
   extends instr_data;
   extends watcher_data;


   output cur_thread_id  :    value reg unsigned<[thread_width]> init 0;
   input threads[max_threads] : temp value thread_type;

   input thread_init;


   // program counter
   output enable_pc ;
   output load_pc_ctsw : temp program_counter_type;
   input watcher_abort;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------


   suspend % when thread_init
      loop
         % setup matrix
         signal result[max_threads/2][thread_width] : temp value unsigned<[
      thread_width]> init 0
         in
            % imit matrix
            for i < max_threads/2 dopar
               run cmp_const [constant (2*i) / id_in1,
                              (2*i+1)  / id_in2;
                        signal result[i][0]/ id_result
                     ]
            end for;
            % perform search over the matrix
            for i < thread_width-1 dopar
               for j <max_threads/4 dopar
                  if j<max_threads/(2*(2**i)) then
                     run cmp [result[2*j][i] /id_in1,
                           result[2*j+1][i] /id_in2;
                           result[j][i+1] /id_result]
                  end if
               end for;
            end for;
            % store the result
            var  id : temp unsigned<[thread_width]> := ?result[0][thread_width-1]
            in
               %compare the current thread_id, if it is the same do nothing
               if not ?threads[id].active then
                  id:=?cur_thread_id
               end if;
               if not ?cur_thread_id=id or watcher_abort then
```

```
                    emit {
                         next ?cur_thread_id <= id, % new current thread
                         ?load_pc_ctsw       <= bin2u(?threads[id].cur_addr),
                         enable_pc %load the pc with the new addr
                        }
                    end if

               end var;
            end signal;
            pause;
         end loop
      when thread_init

   end module

   module cmp_const:

   % helper module for compareing the thread id

      extends thread_data;

      constant id_in1: unsigned<[thread_width]>;
      constant id_in2: unsigned<[thread_width]>;

      input threads[max_threads] :temp value thread_type;
      output id_result : temp value unsigned<[thread_width]>;

   %-----------------------------------------------------------------------------
   %-----------------------------------------------------------------------------

      if ?threads[id_in1].active then
         if ?threads[id_in2].active then
            //both threads are active: take the thread with the higher prio, if prios are
             equal take the thread with the higther id
            if bin2u(?threads[id_in1].prio) > bin2u(?threads[id_in2].prio) then
               emit id_result(id_in1)
            else
               emit id_result(id_in2)
            end if
         else //thread_1 active thread_2 not take thread_1
            emit id_result(id_in1)
         end if
      else // thread_1 not active take thread_2 don't care if active or not
         emit id_result(id_in2)
      end if
   end module

   module cmp:

   % helper module for compareing the thread id

      extends thread_data;

      input threads[max_threads] : temp value thread_type;
      input id_in1         : temp value unsigned<[thread_width]>;
      input id_in2         : temp value unsigned<[thread_width]>;
      output id_result     : temp value unsigned<[thread_width]>;

   %-----------------------------------------------------------------------------
   %-----------------------------------------------------------------------------

      if ?threads[?id_in1].active then
         if ?threads[?id_in2].active then
            //both threads are active: take the thread with the higher prio, if prios are
             equal take the thread with the higther id
            if bin2u(?threads[?id_in1].prio) > bin2u(?threads[?id_in2].prio) then
               emit id_result(?id_in1)
            else
               emit id_result(?id_in2)
            end if
         else //thread_1 active thread_2 not take thread_1
            emit id_result(?id_in1)
         end if
      else // thread_1 not active take thread_2 don't care if active or not
         emit id_result(?id_in2)
      end if


   end module


   module thread_id_relay:

   % solves strange cycle problem for the slow ctws
   % not needed anymore
      extends thread_data;

      input cur_thread_id : temp value unsigned<[thread_width]>;
      output pre_thread_id : temp value unsigned<[thread_width]>;

   %-----------------------------------------------------------------------------
   %-----------------------------------------------------------------------------

      sustain pre_thread_id(?cur_thread_id);

   end module


   module watch_tick_ready:

   % watches if are some running threads, if not emit tick_real_ready

        extends thread_data;
        extends watcher_data;

        input Tick;
        output tick_real_ready;
        input threads[max_threads] : temp value thread_type;
        input watcher_abort;

   %-----------------------------------------------------------------------------
   %-----------------------------------------------------------------------------
```

```
};
    emit tick_real_ready if not ?total_status and not watcher_abort;
suspend
    loop
      signal total_status : temp bool combine or init false
      in
        pause;
        emit (
          for i < max_threads dopar
            ?total_status <= ?threads[i].active
          end for
        end signal;
      end loop
    when not Tick;
  end module
```

D.1.21 join.strl

```
module execute_join :

% test if any child-threads are running, then mark the thread as ready/inactive and
     set the prio parameter if it is not 0
% JOIN Prio
%
% 01010011 pppppppp (16)
%
% module takes 1 tick for computing


   extends thread_data;
   extends watcher_data;

   input    instr_in        :         thread_prio_type;
   inputoutput threads[max_threads] : temp value thread_type;
   input    cur_thread_id   : temp value unsigned<[thread_width]>;
   input    watcher_abort;
   output   enable_pc, thread_ready;


%--------------------------------------------------------------------------
%--------------------------------------------------------------------------

   if instr_in then
      signal childs_running
      in
        emit{
            for i in [1..max_threads-1] dopar
               childs_running if (?threads[i].running) and bin2u((?threads[i].parent))
=?cur_thread_id
            end for
          };
          // emit thread_ready          <= (childs_running);
          pause;

          emit {
             thread_ready            <= pre(childs_running),
             enable_pc               <= not pre(childs_running),
             ?threads[?cur_thread_id].active <= false if pre(childs_running) and not (
watcher_abort),
             ?threads[?cur_thread_id].prio <= ?instr_in if not pre(childs_running) and
bin2u(?instr_in)<>0
          };

           await tick; // delay tick to have equidistant time per instruction
        end signal
      end if

end module
```

D.1.22 prio.strl

```
module execute_prio :

% set prioity of a thread: if Thread<>0 then asign Prio to that thread , else to the
      current thread
% PRIO Thread, PRIO
%
% 01010010 0TTTTTTT pppppppp (24)
% module takes no time for computing

   extends prio_data;
   extends thread_data;

   input instr_in       : temp    prio_type;
   input cur_thread_id  : temp value unsigned<[thread_width]>;
   output enable_pc;
   output threads[max_threads] : temp value thread_type;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   if instr_in then
      var  id : temp unsigned<[thread_width]>
      in
         id:= bin2u(?instr_in.id);
         emit {
              ?threads[?cur_thread_id].prio <= ?instr_in.prio if bin2u(?instr_in.id)
     =0,
              ?threads[id].prio      <= ?instr_in.prio if not bin2u(?instr_in.id)=0
            };
       end var;

       emit enable_pc; // update program_counter
       await 2 tick; // delay 2 tick to have equidistant time per instruction

   end if

end module
```

D.1.23 tickmamager.strl

```
module tick_manager:

% 1. sustains signal Tick over the instance of a tick
% 2. reset tick_len after aTick or a new emit Ticklen
% 3. reset load_count
% 4. decrement tick_len
% 5. wait for tick end to ensure equidistant time per Tick
% 6. sustain tick_warn if the Tick is longer than tick_len

   extends instr_data;
   extends signal_data;


   input    dec_tick_len;
   input    tick_start, reset_load_count;
   input    tick_real_ready;
   input relation tick_manager_rel_1: tick_start # tick_real_ready;
   input    set_tick_len     : temp    instr_data_type;


   output   load_count       : temp    delay_counter_type ;

   output   tick_warn, Tick;
   output   tick_finished;


%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   signal tick_len : reg instr_data_type init 0


   in
      // sustains Tick over the instance
      loop
         abort
            await tick_start;

            sustain Tick;
         when tick_finished
      end loop
      ||
      // reset_tick_len
      loop
         await immediate (tick_finished or set_tick_len);
         emit next tick_len(?set_tick_len);

         pause;
      end loop
      ||
      // reset load_count
      loop
         await reset_load_count;
         emit ?load_count <=u2bin(1,delay_counter_width);
      end loop
      ||
      // decrement tick_len
      loop
         await dec_tick_len;
         if not ?tick_len=0 then
            emit next tick_len(?tick_len-1);
         end if
      end loop
      ||
      //wait for tick end
      loop
         await tick_real_ready and dec_tick_len;
         abort
            loop
               await 5 tick;
               emit dec_tick_len;
            end loop
         when ?tick_len=0;
         emit tick_finished;
      end loop
      ||
      // tick warn
      loop
        await Tick;
         pause;
         if ?tick_len=0 and ?set_tick_len>1 and Tick then
            sustain tick_warn
         end if;

      end loop


   end signal
end module
```

D.1.24 Datentypen

D.1.25 data.strl

```
data constants:
   // UART configuaration (in general fixed)
   constant data_bits       : unsigned <> = 8;    // 8 data bits no parity (8N1)
   constant oversampling    : unsigned <> = 16;   // rate of sampling the input signal
   constant baudrate        : unsigned <> = 115200; // speed
   constant clockrate       : unsigned <> = 100000000; // rate of system clock in Herz
   constant uart_clk_div    : unsigned <> = 2;//clockrate/(baudrate*oversampling)+1;


   // map of the Instruction (in general fixed)
   constant opcode_width    : unsigned <> = 40;
   constant opcode_id_width : unsigned <> = 8;
   constant opcode_signal_width : unsigned <> = 10;
   constant opcode_data_width : unsigned <> = 16;
   constant opcode_addr_width : unsigned <> = 16;
   constant opcode_reg_width : unsigned <> = 10;
   constant opcode_watcher_width : unsigned <> = 6;
   constant opcode_thread_width : unsigned <> = 8; // but max only 7 useable
   constant opcode_prio_width : unsigned <> = 8;

   // KEP described language: true = ESTEREL, false=VHDL (does not make any sense here
      )
   constant kep_lang        : bool     = true;

   ////////////////////////////////////////////////////////////////////
   ////////////////////////// variable configuration //////////////////
   ////////////////////////////////////////////////////////////////////

   // Signal configuration
   //<signal-script>
   constant signal_addr_width : unsigned <> = 5;
   //</signal-script>                                  // max. extreme 9bit -> 512
      Signals
   // DO NOT TOUCH
   constant signal_width    : unsigned <> = signal_addr_width+1; // width of the signal
       including the pre flag

   // Data representation for valued Signals(Integer)
   //<data-script>
   constant data_width      : unsigned <> = 8;
   //</data-script>
   constant instr_data_width : unsigned <> = data_width; // instr_data_width <=
      data_width

   // Instruction configuration
   //<instr-script>
   constant rom_addr_width  : unsigned <> = 7;
   //</instr-script>                                    // max. extreme 16bit ->
      65536 Instructions

   // Instruction trace buffer configuration
   //<trace-script>
   constant trace_buffer_addr_width : unsigned <> = 6;
   //</trace-script>                                          // max. extreme 8bit -> 256
      Instructions(pc-val)


   // Register configuaration
   constant register_addr_width : unsigned <> = 4; //tiny // min. tiny 4bit-> 16
      Register
                                                            // max.extreme 10bit -> 1024
      Register

   // PRE support
   constant pre_support     : bool     = true;

   // delay counter configuration
   constant delay_counter_width : unsigned <> = data_width;     // max.'data_width' (or
       instr_data_width)

   // Thread configuration
   //<thread-script>
   constant thread_width    : unsigned <> = 3;
   //</thread-script>
   constant thread_prio_width : unsigned <> = thread_width+1; // max. 8bit -> 256
      priorities


   // Watcher configuration
   //<watcher-script>
   constant watcher_width   : unsigned <> = 3;
   //</watcher-script>
   constant watcher_counter_width : unsigned <> = delay_counter_width;
   constant twatcher_counter_width : unsigned <> = delay_counter_width;
   // local Watcher configuration
   constant lwatcher_width  : unsigned <> = watcher_width;
   constant lwatcher_counter_width : unsigned <> = watcher_width;



end data

data char_data: // input data of the testdriver

   extends constants;

   type char_type = unsigned <[data_bits]>;
end data

data data_data : // Data representation for valued Signals(Integer)

   extends constants;

   type data_type = unsigned <[data_width]>;
```

```
end data

data instr_data:

  extends constants;
  // start of opcode id
  constant id_start   : unsigned <> = opcode_width - opcode_id_width ; //32 fix

  // frist parameter is SIGNAL
  // SIGNAL only
  constant signal_start : unsigned <> = id_start - opcode_signal_width ; //22 fix
  constant signal_end : unsigned <> = signal_start + signal_width -1; //31 variable
  // SIGNAL followed by DATA
  constant sig_data_start : unsigned <> = signal_start - opcode_data_width ; // 6 fix
  constant sig_data_end : unsigned <> = sig_data_start + instr_data_width -1; //21
    variable
  // SIGNAL followed by REGISTER
  constant sig_reg_start : unsigned <> = signal_start - opcode_reg_width ; //12 fix
  constant sig_reg_end : unsigned <> = sig_reg_start + register_addr_width -1; //21
    variable

  // frist parameter is ADDRESS
  // ADDRESS only
  constant addr_start : unsigned <> = id_start - opcode_addr_width ; //16 fix
  constant addr_end   : unsigned <> = addr_start + rom_addr_width -1; //31 variable
  // ADDRESS followed by SIGNAL
  constant addr_sig_start : unsigned <> = addr_start - opcode_signal_width ; // 6 fix
  constant addr_sig_end : unsigned <> = addr_sig_start + signal_width -1; //15
    variable
  // ADDRESS followed by SIGNAL and WATCHER
  constant addr_sig_w_start : unsigned <> = addr_sig_start - opcode_watcher_width ;
    // 0 fix
  constant addr_sig_w_end : unsigned <> = addr_sig_w_start + watcher_width -1; // 5
    variable
  // ADDRESS followed by ADDRESS
  constant addr_addr_start : unsigned <> = addr_start - opcode_addr_width ; // 0 fix
  constant addr_addr_end : unsigned <> = addr_addr_start + rom_addr_width -1; //15
    variable
  // ADDRESS followed by PRIO
  constant addr_prio_start : unsigned <> = addr_start - opcode_prio_width ; // 8 fix
  constant addr_prio_end : unsigned <> = addr_prio_start + thread_prio_width -1; //15
     variable
   // ADDRESS followed by THREAD_ID
  constant addr_th_start : unsigned <> = addr_start - opcode_thread_width ; // 8 fix
  constant addr_th_end : unsigned <> = addr_th_start + thread_width -1; //15 variable
  // ADDRESS followed by THREAD_ID and PRIO
  constant addr_th_pr_start : unsigned <> = addr_th_start - opcode_prio_width ; // 0
    fix
  constant addr_th_pr_end : unsigned <> = addr_th_pr_start + thread_prio_width -1;
    //7 variable

  // frist parameter is THREAD_ID
  // THREAD_ID only
  constant th_start  : unsigned <> = id_start  - opcode_thread_width ; //24 fix
  constant th_end    : unsigned <> = th_start  + thread_width   -1; //31 variable
  // THREAD_ID followed by PRIO
  constant th_pr_start : unsigned <> = th_start - opcode_prio_width ; //16 fix
  constant th_pr_end : unsigned <> = th_pr_start + thread_prio_width -1; //23
    variable

  // frist parameter is REGISTER
  // REGISTER only
  constant reg_start  : unsigned <> = id_start  - opcode_reg_width ; //22 fix
  constant reg_end    : unsigned <> = reg_start + register_addr_width -1; //31
    variable
  // REGISTER followed by REGISTER
  constant reg_reg_start : unsigned <> = reg_start - opcode_reg_width ; //12 fix
  constant reg_reg_end : unsigned <> = reg_reg_start + register_addr_width -1; //21
    variable
  // REGISTER followed by DATA
  constant reg_data_start : unsigned <> = reg_start - opcode_data_width ; // 6 fix
  constant reg_data_end : unsigned <> = reg_data_start + instr_data_width -1; //21
    vriable


  type instr_type = bool[opcode_width];
     // view of the decoder
     map normal_decoder_instr :instr_type {
        para [0..opcode_width-opcode_id_width-1],
        id  [opcode_width-opcode_id_width..opcode_width-1]
        };
     map spezial_decoder_instr :instr_type {
        first4id [opcode_width-4..opcode_width-1]
        };

  type instr_data_type = unsigned <[instr_data_width]>;

  type rom_addr_type = bool[rom_addr_width];

  type delay_counter_type = bool[delay_counter_width];
  constant reset_delay_counter_const : delay_counter_type = '1;

  type program_counter_type = unsigned <[rom_addr_width]>;

end data

data signal_data:

  extends constants;

  constant max_signals : unsigned <> = 2**signal_addr_width;


  type signal_addr_type = bool[signal_addr_width];
  type signal_value_type = bool[data_width];
  type signal_status_type = bool[max_signals];
  constant reset_signal_const : signal_status_type = '0;
  constant ticklen_signal_addr_const : unsigned<[signal_addr_width]> = 0;
```

```
   type sig_type = bool[signal_width];
      // view of the kep
      map sig_type {
         pre_op[0],
         signal_id[1..signal_width-1]
         };

end data

data register_data:

   extends constants;

   constant max_registers : unsigned <> = 2**register_addr_width;
   constant count_id_const : unsigned<[register_addr_width]> = 0;

   type register_value_type = bool[data_width];

   type register_addr_type = bool[register_addr_width];

end data

data watcher_data :
   extends constants;

  // type preemption_type = enum {WEAK,WEAKI,STRONG,SUSPEND};
   constant preemption_type_width : unsigned <> = 3;
   type preemption_type = bool[preemption_type_width];
   constant STRONG : preemption_type = 'b000 ;
  // constant I_STRONG : preemption_type = 'b001 ;
   constant WEAK : preemption_type = 'b010 ;
   constant I_WEAK : preemption_type = 'b011 ;
   constant SUSPEND : preemption_type = 'b100 ;
   constant I_SUSPEND : preemption_type = 'b101;

   constant max_watcher : unsigned <> = 2**watcher_width;

   type watcher_id_type = unsigned <[watcher_width] >;
   type watcher_type = bool[rom_addr_width+rom_addr_width+signal_width+
     preemption_type_width+delay_counter_width+thread_width+1];
   map watcher_type {
      start_addr [0..
               rom_addr_width - 1],
      end_addr [rom_addr_width..
               rom_addr_width + rom_addr_width - 1],
      isPre    [rom_addr_width + rom_addr_width],
      trigger [rom_addr_width + rom_addr_width + 1..
               rom_addr_width + rom_addr_width + signal_width - 1],
      preemption [rom_addr_width + rom_addr_width + signal_width..
               rom_addr_width + rom_addr_width + signal_width + preemption_type_width
      - 1],
      thread_id [rom_addr_width + rom_addr_width + signal_width +
      preemption_type_width..
               rom_addr_width + rom_addr_width + signal_width + preemption_type_width
    + thread_width - 1]
    // fired   [rom_addr_width + rom_addr_width + signal_width + preemption_type_width
     + thread_width]

      };
end data

data thread_data :
   extends constants;


   constant max_threads : unsigned <> = 2**thread_width;
   constant max_prio : unsigned <> = 2**thread_prio_width;

   type thread_id_type = bool[thread_width];
   type thread_prio_type = bool[thread_prio_width];

   type thread_type = bool[thread_width+thread_prio_width+rom_addr_width+
     rom_addr_width+1+1+1];
   map thread_type {
       parent [0..
               thread_width - 1],
       prio   [thread_width..
               thread_width + thread_prio_width - 1],
       cur_addr [thread_width + thread_prio_width..
               thread_width + thread_prio_width + rom_addr_width - 1],
       end_addr [thread_width + thread_prio_width + rom_addr_width..
               thread_width + thread_prio_width + rom_addr_width + rom_addr_width -
     1],
       running [thread_width + thread_prio_width + rom_addr_width + rom_addr_width],
       active [thread_width + thread_prio_width + rom_addr_width + rom_addr_width+1],
       delayed [thread_width + thread_prio_width + rom_addr_width + rom_addr_width
     +1+1]
      };

end data


data await_data:

   extends constants;

   type sig_await_type = bool[signal_width+1];
   map sig_await_type {
         immed[0],
         pre_op[1],
         signal_id[2..signal_width]
         };

   type await_type = bool[delay_counter_width+1];
   map ext_map : await_type {

         running [0],
         count   [1..
                  delay_counter_width]
```

```
        };
    /* map simp_map : await_type {
        sig_await [0..
                signal_width]

        };
    */
end data
////////////////////////////////////////////////////////////


data sig_data_data :

    extends constants;
    constant sig_data_width : unsigned<> = signal_width+instr_data_width;
    type sig_data_type = bool[sig_data_width];
    map sig_data_type {
        pre_op[0],
        signal_id[1..signal_width-1],
        carried_data[signal_width..sig_data_width-1] };
end data

data sig_reg_data :
    extends constants;
    constant sig_reg_width : unsigned<> = signal_width+register_addr_width;
    type sig_reg_type = bool[sig_reg_width];
    map sig_reg_type {
        pre_op[0],
        signal_id[1..signal_width-1],
        register[signal_width..sig_reg_width-1] };
end data

data sig_addr_data :
    extends constants;
    constant sig_addr_width : unsigned<> = rom_addr_width+signal_width;
    type sig_addr_type = bool[sig_addr_width];
    map sig_addr_type {
        addr[0..rom_addr_width-1],
        pre_op[rom_addr_width],
        signal_id[rom_addr_width+1..sig_addr_width-1]};
end data

data sig_addr_watcher_data :
    extends constants;
    extends sig_addr_data;
    constant sig_addr_watcher_width : unsigned<> = sig_addr_width+watcher_width+
      opcode_id_width-4;
    type sig_addr_watcher_type = bool[sig_addr_watcher_width];
    map sig_addr_watcher_type {
        addr[0..rom_addr_width-1],
        signal_code[rom_addr_width..sig_addr_width-1],
        watcher_id[sig_addr_width..sig_addr_width+watcher_width-1],
        immed[sig_addr_width+watcher_width],
        typ[sig_addr_width+watcher_width+1..sig_addr_watcher_width-1]};
end data

data reg_reg_data :
    extends constants;

    constant reg_reg_width : unsigned<> = 2*register_addr_width;
    type reg_reg_type = bool[reg_reg_width];
    map reg_reg_type {
        source[0..register_addr_width-1],
        target[register_addr_width..reg_reg_width-1]
    };
end data

data reg_data_data :
   extends constants;
    constant reg_data_width : unsigned<> = register_addr_width+instr_data_width;
    type reg_data_type = bool[reg_data_width];
    map reg_data_type {
        target[0..register_addr_width-1],
        carried_data[register_addr_width..reg_data_width-1]
    };
end data

data addr_addr_data :
    extends constants;

    constant addr_addr_width : unsigned<> =2*rom_addr_width ;
    type addr_addr_type = bool[addr_addr_width];
    map addr_addr_type {
        to_addr[0..rom_addr_width-1],
        from_addr[rom_addr_width..addr_addr_width-1]};

end data

data par_data :
    extends constants;

    constant par_width : unsigned<> = rom_addr_width+thread_width+thread_prio_width ;
    type par_type = bool[par_width];
    map par_type {
        addr[0..
            rom_addr_width-1],
        id [rom_addr_width..
            rom_addr_width + thread_width-1],
        prio[rom_addr_width + thread_width..
            rom_addr_width + thread_width + thread_prio_width-1]
        };

end data

data pare_data :
    extends constants;

    constant pare_width : unsigned<> = rom_addr_width+thread_prio_width ;
    type pare_type = bool[pare_width];
    map pare_type {
        addr[0..rom_addr_width-1],
```

```
    prio[rom_addr_width..rom_addr_width+thread_prio_width-1]
      };
end data

data prio_data :
  extends constants;

  constant prio_width : unsigned<> = thread_width+thread_prio_width ;

  type prio_type = bool[prio_width];
  map prio_type {
    id [0..thread_width-1],
      prio[thread_width..thread_width+thread_prio_width-1]
      };
end data
```

D.2 Testdriver

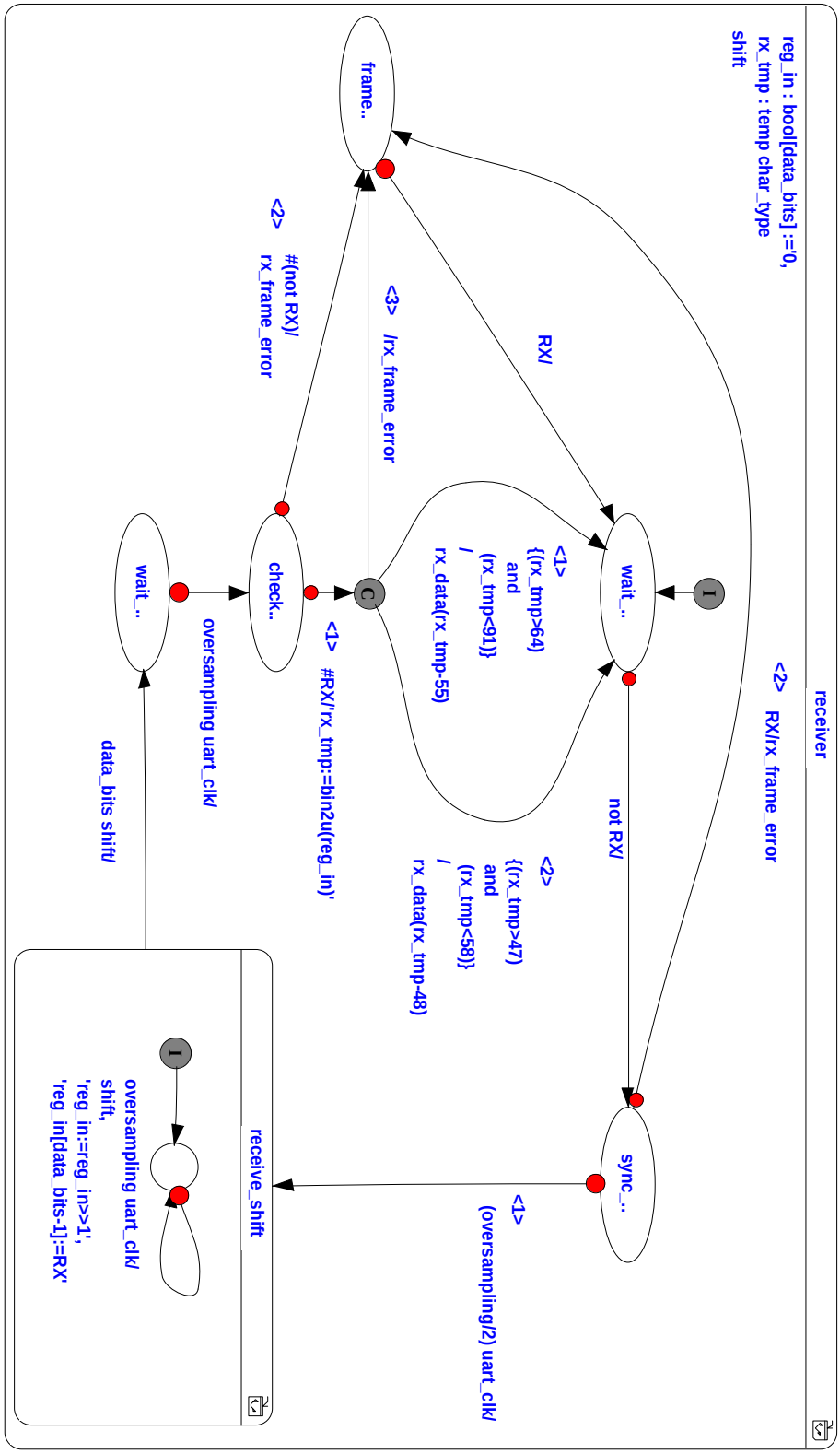


Abbildung D.1: Sendeeinheit des UART

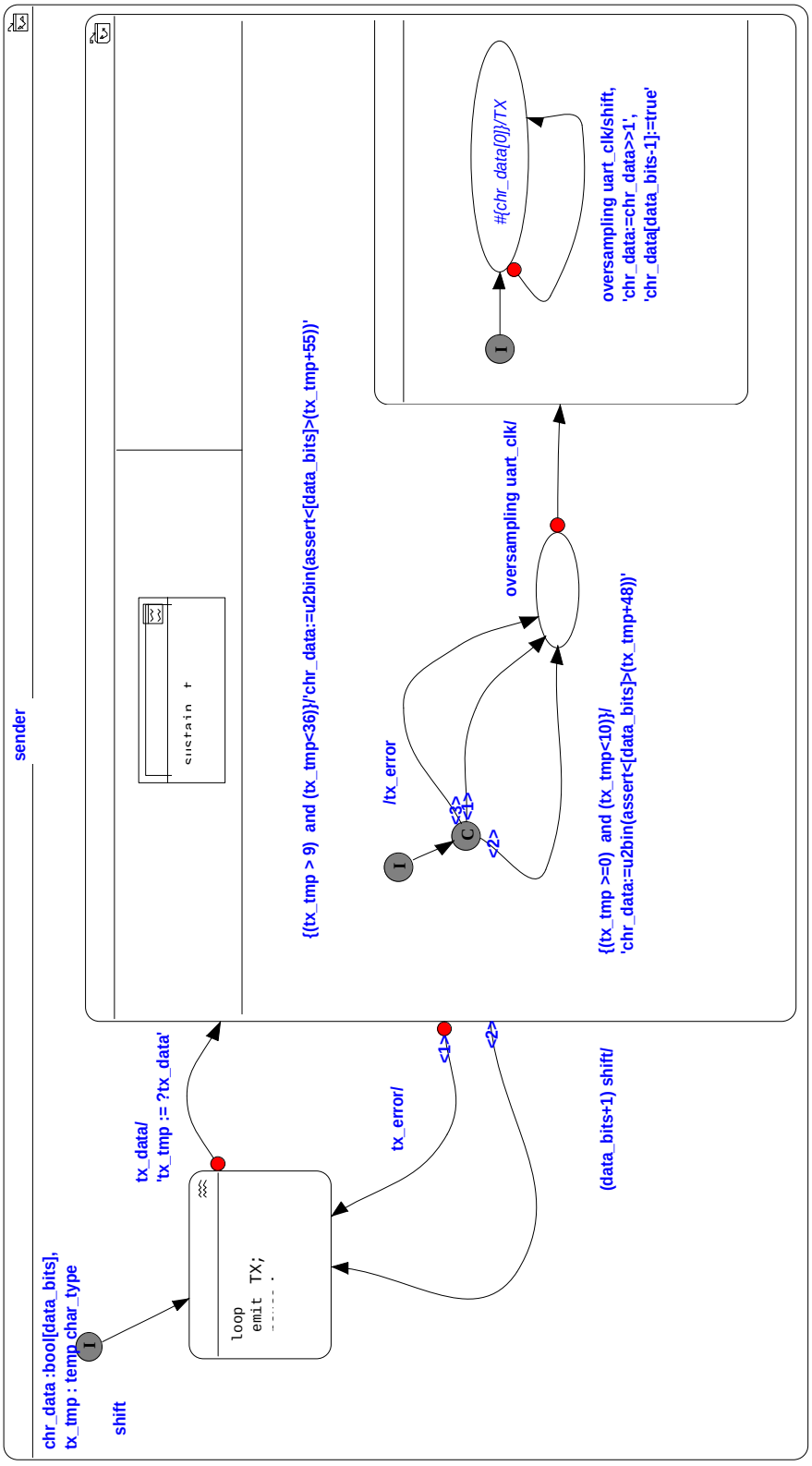


Abbildung D.2: Empfängereinheit des UART

D.2.1 testdriver.strl

```
module testdriver_hardkep:

% wrapper testdriver for hardware kep

   extends char_data;
   extends instr_data;
   extends signal_data;
   extends testdriver_data;


   input RX,Tick;
   output TX,tick_start;
   output no_rx_error,tx_busy,no_tx_error,reset_kep;


   /*// valued signal ram
   output signal_ram_addr : temp value signal_addr_type ;
   input signal_val_from_ram : temp value signal_value_type;
   output signal_val_to_ram : temp value signal_value_type;
   output wea_signal;
   */

   // instruction rom
   output instr_to_rom  : temp value instr_type;
   input instr_from_rom : temp value instr_type;
   inputoutput rom_addr :temp rom_addr_type ;
  // input rom_addr_in    : rom_addr_type init '0;
   output wea_instr;

   // execution trace
   output trace_buffer_addr : temp value trace_buffer_addr_type ;
   output pc_val_to_buffer : temp value rom_addr_type ;
   input pc_val_from_buffer : temp value rom_addr_type;
   output wea_trace;

   // signal status
   output env_signals   : temp value signal_status_type ;
   input pre_signals    : temp value signal_status_type ;
   input instr_count :temp value unsigned<[trace_buffer_addr_width]>;

%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------

  signal
   rx_data              : temp char_type,
   tx_data              : temp char_type,
   rx_frame_error,
   tx_error,read_error,V,W,T,I,O,N,M,L
   in
      run uart
      ||
      loop
         abort
            loop
               await rx_data;
               %  integrate the decode for optimization
               emit {
                    //D <= ?rx_data = 13,
                    //E <= ?rx_data = 14,
                    //F <= ?rx_data = 15,
                    //G <= ?rx_data = 16,
                    I <= ?rx_data = 18,
                    L <= ?rx_data = 21,
                    M <= ?rx_data = 22,
                    N <= ?rx_data = 23,
                    O <= ?rx_data = 24,
                    //P <= ?rx_data = 25,
                   // R <= ?rx_data = 27,
                    T <= ?rx_data = 29,
                    V <= ?rx_data = 31,
                    W <= ?rx_data = 32
                };

               run execute_task;

            end loop
           when reset_kep
         end loop
        ||
       run save_trace
        ||
        run error_mod
        ||
        run run_reset


  end signal;
end module

module testdriver_softkep:

% wrapper testdriver for software kep

   extends char_data;
   extends instr_data;
   extends signal_data;
   extends testdriver_data;

   input Tick;
   input rx_raw_data    : temp char_type;
   output tx_data       : temp char_type;
   output tick_start;
   output no_rx_error,tx_busy,no_tx_error,reset_kep;
```

```
   /*// valued signal ram
   output signal_ram_addr : temp value signal_addr_type ;
   input signal_val_from_ram : temp value signal_value_type;
   output signal_val_to_ram : temp value signal_value_type;
   output wea_signal;
   */

   // instruction rom
   output instr_to_rom  : temp value instr_type;
   input instr_from_rom : temp value instr_type;
   inputoutput rom_addr     :temp rom_addr_type ;
  // input rom_addr_in    : rom_addr_type init '0;
   output wea_instr;

   // execution trace
   output trace_buffer_addr : temp value trace_buffer_addr_type ;
   output pc_val_to_buffer : temp value rom_addr_type ;
   input pc_val_from_buffer : temp value rom_addr_type;
   output wea_trace;

   // signal status
   output env_signals   : temp value signal_status_type ;
   input pre_signals    : temp value signal_status_type ;
   input instr_count :temp value unsigned<[trace_buffer_addr_width]>;

   signal
      rx_data              : temp char_type,
      rx_frame_error,
      tx_error,read_error,V,W,T,I,O,N,M,L
   in
      run uart_dummy
      ||
      loop
         abort
           loop
             await rx_data;
             %   integrate the decode for optimization
             emit {
                 // D <= ?rx_data = 13,
                 // E <= ?rx_data = 14,
                 // F <= ?rx_data = 15,
                 // G <= ?rx_data = 16,
                  I <= ?rx_data = 18,
                  L <= ?rx_data = 21,
                  M <= ?rx_data = 22,
                  N <= ?rx_data = 23,
                  O <= ?rx_data = 24,
                 // P <= ?rx_data = 25,
                 // R <= ?rx_data = 27,
                  T <= ?rx_data = 29,
                  V <= ?rx_data = 31,
                  W <= ?rx_data = 32
              };

             run execute_task;
           end loop
           when reset_kep
      end loop
      ||
      run save_trace
      ||
      run error_mod
      ||
      run run_reset


  end signal;
end module


module decode_char:

% wait for incomming char and decode it

   extends char_data;

   input rx_data : temp char_type;

   output V,W,D,G,T,P,I,O,N,M,L,E,F;
   output relation decode_char_rel: V # W # D # G # T # P # I # O # N # M # L # E # F;

   await rx_data;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   emit {
      D <= ?rx_data = 13,
      E <= ?rx_data = 14,
      F <= ?rx_data = 15,
      G <= ?rx_data = 16,
      I <= ?rx_data = 18,
      L <= ?rx_data = 21,
      M <= ?rx_data = 22,
      N <= ?rx_data = 23,
      O <= ?rx_data = 24,
      P <= ?rx_data = 25,
     // R <= ?rx_data = 27,
      T <= ?rx_data = 29,
      V <= ?rx_data = 31,
      W <= ?rx_data = 32
   }

end module

module execute_task:

% activate module for the action

   extends char_data;
   extends instr_data;
```

```
    extends signal_data;
    extends testdriver_data;

    input V,W,T,I,O,N,M,L, tx_busy,Tick;
    input relation exe_task_rel: V # W # T # I # O # N # M # L ;
    input rx_data       : temp char_type;
    /*// signal ram
    input signal_val_from_ram : temp value signal_value_type;
    output signal_val_to_ram : temp value signal_value_type;
    output signal_ram_addr : temp value signal_addr_type;
    output wea_signal;
    */
    output tx_data      : temp char_type;
    output instr_to_rom : temp value instr_type;
   // input instr_from_rom : temp value instr_type;
    output rom_addr     : temp rom_addr_type;

    input pre_signals   : temp value signal_status_type;
    output env_signals  : temp value signal_status_type;
    output trace_buffer_addr : temp value trace_buffer_addr_type;
    input pc_val_from_buffer : temp value rom_addr_type;
  // output pc_val_to_buffer : temp value rom_addr_type;
    output wea_instr,tick_start,read_error;
    input instr_count :temp value unsigned<[trace_buffer_addr_width]>;

%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
    run write [ W / enable ]
    ||
    run verify [V / enable ]
    /*||
    run send_instr_rom [E / enable ]
    ||
    run send_signal_ram [F / enable ]
    ||
    run sel_signal [D / enable ]
    ||
    run set_value [G / enable]
    ||
    run send_value [P / enable ]
    */
    ||
    run run_tick [T / enable ]
    ||
    run harvest_sig_st [O / enable ]
    ||
    run seed_sig_st [I / enable ]
    ||
    run send_tick_len[L /enable]
    ||
    run info[N / enable]
    ||
    run send_trace[M / enable]
end module
```

D.2.2 error.strl

```
module error_mod:

% propagate read or write error to the env.

   input rx_frame_error,tx_error,read_error;
   output no_rx_error,no_tx_error;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------
                                                                              10
      await rx_frame_error or read_error;
      sustain no_rx_error;
      ||
      await tx_error;
      sustain no_tx_error;

end module
```

D.2.3 verify.strl

```
module verify:

% send verification sting to the host for testing the communication

   extends char_data;

   input enable, tx_busy;
   output tx_data : temp char_type;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   if enable then
     //send '0123456789ABCDEF'
       repeat i:=16 times
           await not tx_busy;
           pause;
           emit tx_data(16-i);

       end repeat;
       // send 'X'
       await not tx_busy;
       pause;
       emit tx_data(33);
   end if
end module
```

D.2.4 reset.strl

```
module run_reset:

% reset the KEP

   extends char_data;

   input tx_busy;
   input rx_data : temp char_type;

   output tx_data : temp char_type;
   output reset_kep;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   loop
      await rx_data;

      if ?rx_data=27 then // reset the KEP

         //abort // sustain the signal for more then one cycle

            emit reset_kep;
            pause; // reset takes a cycle

         //when 5 times tick;

         // send 'X' to indicate the end of reset
         await not tx_busy;
         pause;
         emit tx_data(33);
      end if
   end loop
end module
```

D.2.5 info.strl

```
module info:

%send configuation of the on-board kep
% quick hack

   extends char_data;
   extends register_data;
   extends constants;
   extends testdriver_data;

   input enable, tx_busy;
   output tx_data : temp char_type;



%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   if enable then
      signal
         send_two_char_value : temp two_chars_type,
         send_two_chars_done,info_finished
      in
         run send_two_char
         ||

         // send KEP language and pre support
         await not tx_busy;
         pause;
         emit tx_data(bin2u(mcat(kep_lang,'0,'0,pre_support)));

         // send Data path width
         emit send_two_char_value(extend(u2bin(data_width),2*hex_char));
         await send_two_chars_done;


         // send maximum number of signals
         var max_signals_val : testdriver_sig_addr_type := extend(u2bin(max_signals),
      testdriver_sig_addr_width)
         in
            repeat testdriver_sig_addr_width/hex_char times
               await not tx_busy;
               pause;
               emit tx_data(bin2u(max_signals_val.msb));
               max_signals_val:=max_signals_val<<hex_char;
            end repeat
         end var;

         //send maximum number of threads
         emit send_two_char_value(extend(u2bin(2**thread_width),2*hex_char));
         await send_two_chars_done;

         //send maximum number of watcher
         emit send_two_char_value(extend(u2bin(2**watcher_width),2*hex_char));
         await send_two_chars_done;

         //send counter width of watcher
         emit send_two_char_value(extend(u2bin(watcher_counter_width),2*hex_char));
         await send_two_chars_done;

         //send maximum number of local watcher
         emit send_two_char_value(extend(u2bin(2**lwatcher_width),2*hex_char));
         await send_two_chars_done;

         //send counter width of local watcher
         emit send_two_char_value(extend(u2bin(lwatcher_counter_width),2*hex_char));
         await send_two_chars_done;

         //send counter width of thread watcher
         emit send_two_char_value(extend(u2bin(twatcher_counter_width),2*hex_char));
         await send_two_chars_done;

         //send delay counter width
         emit send_two_char_value(extend(u2bin(delay_counter_width),2*hex_char));
         await send_two_chars_done;

         //send pio value width
         emit send_two_char_value(extend(u2bin(thread_prio_width),2*hex_char));
         await send_two_chars_done;

         // send maximum number of registers
         var max_reg_val : testdriver_sig_addr_type := extend(u2bin(max_registers),
      testdriver_sig_addr_width)
         in
            repeat testdriver_sig_addr_width/hex_char times
               await not tx_busy;
               pause;
               emit tx_data(bin2u(max_reg_val.msb));
               max_reg_val:=max_reg_val<<hex_char;
            end repeat
         end var;

         //send max tick length
         var max_tick_len_val : testdriver_instr_addr_type := extend(u2bin(2**
      trace_buffer_addr_width),testdriver_instr_addr_width)
         in
            repeat 4 times
               await not tx_busy;
               pause;
               emit tx_data(bin2u(max_tick_len_val.msb));
               max_tick_len_val:=max_tick_len_val<<hex_char;
            end repeat
         end var;
```

```
                                    input tx_busy,info_finished;
                                    input send_two_char_value : temp two_chars_type;
                                    output send_two_chars_done;
                                    output tx_data      : temp char_type;
                                    abort
                                       loop
                                          await immediate send_two_char_value;
                                          var send_val_temp : two_chars_type := ?send_two_char_value
                                          in
                                             repeat 2 times
                                                await not tx_busy;
                                                pause;
                                                emit tx_data(bin2u(send_val_temp.msb));
                                                send_val_temp:=send_val_temp<<hex_char;
                                             end repeat
                                          end var;
                                          emit send_two_chars_done;
                                       end loop;
                                    when info_finished
                                    end module
      //send Instruction memory addr width
      emit send_two_char_value(extend(u2bin(rom_addr_width),2*hex_char));
      await send_two_chars_done;

   // send 'X' to denote end of msg
   await not tx_busy;
   pause;
   emit tx_data(33);
   emit info_finished;
 end signal
 end if
end module

module send_two_char:

extends char_data;
extends testdriver_data;
```

D.2.6 runtick.strl

```
module run_tick:

% run a Tick and wait for completion of the macro tick

   extends char_data;
   extends testdriver_data;
   extends instr_data;

   constant reset_trace_addr_const : trace_buffer_addr_type='0;

   input enable, tx_busy,Tick;
   output tx_data : temp char_type;
   output tick_start;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   if enable then

     emit tick_start;
     await not Tick and pre(Tick); //tick finished
     pause;
     await not tx_busy;
     pause;
     emit tx_data(33);
   end if
end module
```

D.2.7 reset.strl

```
module run_reset:

% reset the KEP

   extends char_data;

   input tx_busy;
   input rx_data : temp char_type;

   output tx_data : temp char_type;
   output reset_kep;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   loop
      await rx_data;

      if ?rx_data=27 then // reset the KEP

         //abort // sustain the signal for more then one cycle

            emit reset_kep;
            pause; // reset takes a cycle

         //when 5 times tick;

         // send 'X' to indicate the end of reset
         await not tx_busy;
         pause;
         emit tx_data(33);
      end if
   end loop
end module
```

D.2.8 harvestsigst.strl

```
module harvest_sig_st:

% send the status of the signals to the host

   extends signal_data;
   extends testdriver_data;
   extends char_data;

   input enable,tx_busy;
   input pre_signals   :  temp value signal_status_type;
   output tx_data : temp char_type;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------


   if enable then
      var signal_status : signal_status_type:=?pre_signals
       in
         signal_status:=signal_status>>1;
         repeat (2**signal_addr_width)/hex_char times
             await not tx_busy;
             pause;
             emit tx_data(bin2u(signal_status.lsb));
             signal_status:=signal_status>>hex_char;
          end repeat
      end var;
      await not tx_busy;
      pause;
      emit tx_data(33);
   end if

end module
```

D.2.9 seedsigst.strl

```
module seed_sig_st:

% set the env. status of the signals

   extends signal_data;
   extends testdriver_data;
   extends char_data;

   input rx_data : temp char_type;

   input enable;
   output env_signals : temp value signal_status_type ;
   output read_error;

%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------


   // set the status of the signals
   if enable then

   signal T[max_signals/hex_char]: char_type init 0,
        O: unsigned<[max_signals]> combine +
   //   K[max_signals/hex_char]: unsigned<[max_signals]>
   in

   //read
      var i : unsigned<max_signals/hex_char> := 0
      in
         trap finished in
            loop
               await rx_data;
               if
                  case ?rx_data=33 do exit finished
                  case ?rx_data<16 do
                     emit T[i](?rx_data);
                     i:=sat<max_signals/hex_char>(i+1);
                  default do emit read_error
               end if
            end loop;
         end trap;



      // set the signals
         emit {
            for i< max_signals/hex_char dopar
               // ?K[i] <= sat<[4*N]>(?T[i]*(2**(4*i))),
               ?O <= sat<[max_signals]>(?T[i]*(2**(hex_char*i)))
            end for
            };

          emit ?env_signals <= u2bin(?O)<<1;
         end var;
   end signal
   end if
end module
```

D.2.10 write.strl

```
module write:

% receive instructions (the program) and write it to the instr_rom
% read RomAddr then read 10 char and write the value to the ROM


   extends char_data;
   extends instr_data;

   output tx_data : temp char_type;
   input rx_data  : temp char_type;
   input enable,tx_busy;
   output instr_to_rom : temp value instr_type;
   output rom_addr : temp rom_addr_type;
   output wea_instr :reg ;


%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   if enable then

   signal read_error,
        instr_rec : temp instr_type,
        rom_rec : temp rom_addr_type
   in
      var inst_val_first : instr_type :='0,
          inst_val_check : instr_type :='1,
          addr_val_first : rom_addr_type :='0,
          addr_val_check : rom_addr_type :='1
      in
         trap finished in
            loop
               trap read_error_t in
                  loop
                     // read address and instruction (first run)
                     await rx_data;
                     if
                        case ?rx_data = 6 do [
                              run get_instr_data;       // host sent '6' to denote
   start of instruction address pair
                              if instr_rec and rom_rec then
                                 inst_val_first:=?instr_rec;
                                 addr_val_first:=?rom_rec;
                              else emit read_error;
                              end if;
                                    ]
                        case ?rx_data = 33 do exit finished // host sent 'X' to denote
    end of transfer
                        default do emit read_error      // host sent wrong char
                     end if;
                     // read address and instruction (check run)
                     await rx_data;
                     if ?rx_data = 7 then [              // host sent '7' to denote
   check/repeat of instruction address pair
                        run get_instr_data;
                        if instr_rec and rom_rec then
                           inst_val_check:=?instr_rec;
                           addr_val_check:=?rom_rec;
                        else emit read_error;
                        end if;
                                    ]
                     else
                        emit read_error                 // host sent wrong char
                     end if;

                     if (inst_val_first=inst_val_check) // first and check instruction
    address pair does match -> write it to inst_rom
                        and
                       (addr_val_first=addr_val_check)
                     then [
                        emit rom_addr (addr_val_check);
                        emit instr_to_rom(inst_val_check);
                        emit next wea_instr;
                         ]
                     else emit read_error               // first and check instruction
   address pair does not match -> notify host
                     end if;
                     if read_error then
                        exit read_error_t;
                     end if
                  end loop
                  ||
                    await read_error;
                    pause;
                    sustain read_error;

               end trap; //read_error_t
               // send 'X' to notify host 'read error'
               await not tx_busy;
               pause;
               emit tx_data(33);
            end loop
         end trap // finished
      end var
   end signal
   end if
end module

module get_instr_data:


   extends char_data;
```

```
extends instr_data;
extends testdriver_data;

input  rx_data  : temp char_type;
output rom_rec : temp rom_addr_type;
output instr_rec : temp instr_type;
output read_error;

var inst_val : instr_type :='0,
    addr_val : testdriver_instr_addr_type :='0
in

   // read address
   repeat opcode_addr_width/hex_char times
      await rx_data;
      if ?rx_data<16 then
         addr_val:=addr_val<<hex_char;
         addr_val.lsb:=(u2bin(assert<[hex_char]>(?rx_data)));
      else emit read_error
      end if
   end repeat;

   // read instruction
   repeat opcode_width/hex_char times
      await rx_data;

         if ?rx_data<16 then
            inst_val:=inst_val<<hex_char;
            inst_val.lsb:=(u2bin(assert<[hex_char]>(?rx_data)));
         else emit read_error
         end if
      end repeat;

      emit rom_rec(addr_val.addr);
      emit instr_rec(inst_val);

   end var
end module
```

D.2.11 savetrace.strl

```
module save_trace:

% save every pc while tick

   extends testdriver_data;
   extends instr_data;

   output wea_trace: reg;
   output trace_buffer_addr : temp value trace_buffer_addr_type;
   output pc_val_to_buffer : temp value rom_addr_type;
   input rom_addr          : temp rom_addr_type;
   input Tick;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   loop
      abort
         await Tick;
         var addr : trace_buffer_addr_type :='0
         in

            every immediate rom_addr do
               emit pc_val_to_buffer(?rom_addr);
               emit trace_buffer_addr(addr);
               emit next wea_trace;
               addr:=u2bin(sat<[trace_buffer_addr_width]>(bin2u(addr)+1))
            end every
         end var
      when pre(Tick) and not Tick
   end loop
end module
```

D.2.12 sendticklen.strl

```
module send_tick_len:

% send tick_len of the last tick

   extends char_data;
   extends signal_data;
   extends testdriver_data;

   output tx_data  : temp char_type;

   input enable,tx_busy;
   input instr_count : value temp unsigned<[trace_buffer_addr_width]>;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   if enable then
      var max_tick_len : trace_buffer_addr_type :=u2bin(?instr_count)
      in
         repeat 4 times
            await not tx_busy;
            pause;
            emit tx_data(bin2u(max_tick_len.lsb));
            max_tick_len:=max_tick_len>>hex_char;
         end repeat
      end var;
      // send 'X'
      await not tx_busy;
      pause;
      emit tx_data(33)
   end if
end module
```

D.2.13 sendinstrrom.strl

```
module send_instr_rom:

% sends the whole instr_rom for debugging

   extends char_data;
   extends instr_data;
   extends testdriver_data;

   output tx_data : temp char_type;

   input enable,tx_busy;
   input instr_from_rom : temp value instr_type;
   output rom_addr : temp rom_addr_type;

%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------

   if enable then
      repeat i:=2**rom_addr_width-1 times
         emit rom_addr(u2bin(2**rom_addr_width-1-i));
         pause;
         var instr : instr_type:=?instr_from_rom
         in
            repeat opcode_width/hex_char times
               await not tx_busy;
               pause;
               emit tx_data(bin2u(instr.lsb));
               instr:=instr>>hex_char;
            end repeat
         end var;
      end repeat
   end if

end module
```

D.2.14 sendtrace.strl

```
module send_trace:

% send execution trace to the host

   extends char_data;
   extends testdriver_data;

   output tx_data : temp char_type;

   input instr_count   : temp value unsigned<[trace_buffer_addr_width]>;
   output trace_buffer_addr : temp value trace_buffer_addr_type;
   input pc_val_from_buffer : temp value rom_addr_type;


   input enable,tx_busy;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   if enable then
      var end_trace : unsigned<[trace_buffer_addr_width]>
      in
         end_trace :=?instr_count;
         repeat i:=end_trace times

            emit trace_buffer_addr(u2bin(end_trace-i));
            await not tx_busy;
            pause;
            pause;
            var pc_val : rom_addr_type:=?pc_val_from_buffer
            in
               repeat opcode_addr_width/hex_char times
                  emit tx_data(bin2u(pc_val.lsb));
                  pc_val:=pc_val>>hex_char;
                  await not tx_busy;
                  pause;
               end repeat
            end var
         end repeat
      end var;
      // send 'X'
      await not tx_busy;
      pause;
      emit tx_data(33)


   end if

end module
```

D.2.15 sendvalue.strl

```
module send_value:

% send value of the selected signal to host

   extends char_data;
   extends signal_data;
   extends testdriver_data;

   output tx_data : temp char_type;

   input enable,tx_busy;
   input signal_val_from_ram : temp value signal_value_type;

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

   if enable then
      var signal_val : signal_value_type:=?signal_val_from_ram
      in
         repeat data_width/hex_char times
            await not tx_busy;
            pause;
            emit tx_data(bin2u(signal_val.msb));
            signal_val:=signal_val<<hex_char;
         end repeat
      end var;
      // send 'X'
      await not tx_busy;
      pause;
      emit tx_data(33)
   end if

end module
```

D.2.16 setvalue.strl

```
module set_value:

% set the value of the selected signal

   extends char_data;
   extends signal_data;
   extends testdriver_data;

   input rx_data : temp char_type;

   input enable;
   output signal_val_to_ram : temp value signal_value_type;
   output read_error;
   output wea_signal ;

%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------

   if enable then
      var signal_val : signal_value_type:='0
      in
         trap finished in
            loop
               await rx_data; // wait for incomming char
                  if
                  case ?rx_data=33 do exit finished // exit if 'X' (end of msg)
                  case ?rx_data<16 do
                     signal_val:=signal_val>>hex_char;
                     signal_val.msb:=u2bin(assert<[hex_char]>(?rx_data));
                  default do emit read_error
                  end if
            end loop
         end trap;
         emit signal_val_to_ram(signal_val);
         emit wea_signal;
         pause;

      end var
   end if

end module
```