-

Asynchrone Prozessoren

SE Seminar aus Informatik

Martin Hutle
881/9626133

3. Februar 2000

Inhaltsverzeichnis

Teil I
Einleitung

1 Was ist Asynchrone Logik?

Beim herkömmlichem VLSI-Entwurf signalisiert eine zentrale Taktinstanz, die sog. *clock*, wann das Ergebnis eines Logikblocks in das Ergebnisregister geschrieben wird und somit den Nachfolgeblöcken zur Verfügung steht. Asynchrone Schaltungen kommen ohne zentralen Takt aus und synchronisieren die Weitergabe der Ergebnisse an die Folgeblöcke selbständig.

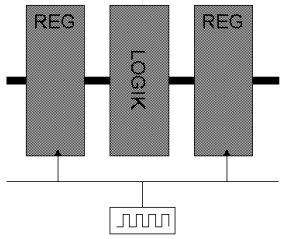

Abbildung 1: Schemata eines synchronen Systems

1.1 Historisches

Die Geschichte asynchroner Logik beginnt schon sehr früh, genau genommen sogar vor der Geschichte der „synchronen“, getakteten Logik: schon in den 50er Jahren wurden die ersten Forschungen in diesem Gebiet unternommen. John von Neumann entwarf zu dieser Zeit in Priceton den IAS und an der University of Illinois wurde der ORDVAC gebaut. Pioniere der Asynchronen Logik waren David Muller (nach ihm ist auch das Muller-C-Element [s. Kapitel 3.2] benannt) an der University of Illinois, Al Davis an der University of Utha und Jack Dannis am MIT.

In den Sechzigern entwickelte Ivan E. Sutherland ein asynchrones Design für einen Graphikprozessor der speziell das Problem des *„clippings“* in der Computergraphik mittels einer asynchronen, elastischen Pipeline implementierte. Später hat Sutherland sein Konzept zu den sog. *micropipelines* verallgemeinert [1] (s.a. Kapitel 3). Weitere wichtige Forschungen im Bereich der *micropipelines* wurden von Charles Molnar an der Washington University in den Siebzigern durchgeführt.

Bis zu diesem Zeitpunkt basierten alle Forschungen auf den *micropipelines* , die jedoch den Nachteil

haben, nicht wirklich unabhänig von den Verzögerungszeiten der Gatter zu sein. Sie brauchen zur Synchronisation ein Verzögerungsglied, das mindestens die selbe Verzögerung (*delay*) wie die zu synchronisierende Pipelinestufe hat. Ganz ohne Verzögerungsglied kommen Verfahren wie das double-rail-coding oder ähnliche aus. Diese Ideen entstanden erst später, Artikel über diese Technik stammen meist aus den Achzigern oder Neunzigern. Die ersten praktischen Erfolge konnte die Firma Theseus Logic Inc. unter Karl M. Fant mit der NULL-Convention-Logic seit 1992 erzielen.

Fertige Prozessoren gibt es von der Manchester University und von der Tokio University of Technologies mit den Produkten AMULET3 und TITAC-2, wobei ersterer schon bald mit Hilfe der Firma Hagenuk auf den Markt kommen soll.

1.2 Definitionen

In der Literatur existieren in Zusammenhang mit asynchroner Logik verschiedene Begriffe die manchmal auch mit unterschiedlichen Bedeutungen belegt sind. Hier sollen die wichtigsten Begriffe definiert werden wie sie im Folgenden verwendet werden:

synchronous system System, das mit einem zentralen Takt betrieben wird.

asynchronous system System, das ohne Takt auskommt.

delay insensitive logic Logik, die ohne Berücksichtigung der Gatterlaufzeit funktioniert.

self timed circuits Asynchrone Schaltungen, deren korrekte Funktion weder von der Geschwindigkeit ihrer Komponenten noch von den Signalverzögerungszeiten in den Leitungen abhängen. [9]

self timed system Ein System, das entweder wiederum aus *self timed systems* oder aus *self timed circuits* besteht, die sich untereinander selbst synchronisieren.

internally clocked circuits Schaltungen, die zwar intern mit einem Taktgeber arbeiten, aber sich an ihren Ausgängen wie ein asynchroner Baustein verhalten. [4]

temporal logic Erweiterung der boolschen Logik, die Zeitbedingungen modellieren kann. So existieren z.B. die Operatoren *henceforth* oder *eventually*. Mehr dazu findet sich z.B. unter [14].

2 Warum eigentlich asynchron?

2.1 Die Grenzen synchroner Schaltungen

Heutzutage funktionieren nicht nur alle gängigen Prozessoren mittels eines zentralen Taktgebers, die Taktfrequenz ist auch vielfach (wenn vielleicht auch unberechtigt) eine Maßzahl für die Leistung eines Microprozessors geworden. Umso schwerer ist auch aus Marketingsicht der Schritt zu einer Technologie, die „keine Megahertz“ hat. Auch sämtliche Entwicklungstools, Desingerregeln und Erfahrungen sind auf den synchronen Fall zugeschnitten und sind in der neuen asychronen Welt nur schwer zu gebrauchen.

Dennoch ist die Suche nach neuen Technologien wohl nicht vermeidbar. Wenn sich die jetzige Entwicklung in der Halbleitertechnik so weiterentwickelt sähe die Zukunft so aus:

Jahr	2007	2012
Strukturgröße (nm)	100	50
Packungsdichte	50×10^6	180×10^6
Transistoren/cm^2	500×10^6	135×10^9
Chipfläche (cm^2)	10	750
Gatterlaufzeit (ps)	10	10
Taktfrequenz (GHz)	3	10

Signale legen auf Leitungen ca. 1 mm in 100 ps zurück. Um einen Chip mit 30 mm Seitenlänge zu durchqueren bräuchten sie bei 3 GHz also 10 Takte, im zweiten Fall (mit 270 mm Seitenlänge und 10 GHz) sogar 270 (!) Taktzyklen. Das solche Chips mit einer zentralen Uhr nicht mehr betrieben werden können ist wohl offensichtlich (Beispiel aus [3]).

Weiters muß sich der Systemtakt nach der langsamsten Komponente richten, diese bremst schnellere Operationen aus. Alle Komponenten schalten im Rythmus des Takts, ob sie nun etwas sinnvolles tun oder nicht. Das wiederum erhöht die Verlust-

leistung, die (in Form von Wärme) zusammen mit den immer kleineren Strukturen für einen erhöhten Wiederstand sorgt, welcher wiederum die Geschwindigkeit der Signale bremst. Zu allem Übel führen die immer höheren Taktfrequenzen zu einer verstärkten hochfrequenten Abstrahlung, welche andere technische Systeme in ihrer Funktion behindern kann.

Ein weiteres großes Problem liegt im Design heutiger Prozessoren. Schon heute dauert die Entwicklung für einen modernen Prozessor circa 3 Jahre. Da sich derzeit allein durch die Verbesserung der Halbleitertechnik schon eine Erhöhung der Taktrate um den Faktor 2 in 18 Monaten erfolgt, muß, um einen schnelleren Chip zu erreichen, zu Beginn des Designs eine Architektur angestrebt werden, die mehr als das vierfache der aktuellen Prozessorgeschwindigkeit zu Beginn der Entwicklung leistet. Nicht unbedingt eine leichte Aufgabe, und oft müssen von den angestrebten Zielen viele Abstriche gemacht werden. Asynchrone Schaltungen synchronisieren sich selbst mit ihrer Umgebung und erleichtern somit im Normalfall den Entwurf, da sich die Funktionsüberprüfungen auf einzelne Teile beschränken, die dann, ohne globale Signalpfade o.ä. berücksichtigen zu müssen, aneinandergefügt werden können.

Alles in allem ergeben sich genug Gründe für die Suche nach einer neuen Technik, die die genannten Probleme vermeidet. Die Asynchrone Logik könnte eine Möglichkeit dazu sein.

2.2 Probleme asynchroner Schaltungen

Das Hauptproblem bei Schaltungen ohne Takt ist, daß Ergebnisse auf verschiedenen Leitungen unterschiedlich schnell fertig werden können, und somit für eine gewisse Zeit ein falsches Ergebnis an den Eingängen der Nachfolgestufe anliegen kann. Weiters ist nicht bekannt, wie lange ein Ergebnis gehalten werden muß, bevor weitergerechnet werden kann. Bei synchronen Schaltungen übernimmt diese Funktion der zentrale Schiedsrichter Takt, ist kein solcher vorhanden, müssen sich die Funktionsblöcke selbst synchronisieren.

2.3 Lösungsansätze

Grob gesehen lassen sich sämtliche Lösungsansätze zur Realisierung asynchroner Schaltungen in zwei große Gruppen unterteilen: In Lösungen in denen die eigentliche Funktion - also im Wesentlichen der Datenpfad - unverändert in konventioneller Logik ausgeführt und durch eine spezielle Logik zum synchronisieren der einzelnen Funktionsblöcke ergänzt wird, und in Lösungen, in denen die timing-Information in die Logik einbezogen wird.

Die wichtigsten Vertreter der ersten Gruppe sind die Micropipelines von I. E. Sutherland. Für diese Technik existieren auch schon die meisten praktischen Realisierungen bis hin zu vertriebsreifen Prozessoren. Weniger Verbreitung finden Ideen wie Laufzeitabschätzungen oder Verfahren, die das Ergebnis einer Berechnung durch elektrotechnische Messungen feststellen (*CSCD, current sensing completetion detection.*

Für die zweite Gruppe gibt es verschiedene Verfahren die alle auf dem gleichen Prinzip basieren, nämlich eine mehrwertige (meist dreiwertige) Logik einzuführen, deren Werte die Information über die Vollständigkeit der Information mit codiert haben. Sie unterscheiden sich nur durch die verschiedenen technischen Realisierungen im Bezug auf Codierung in binärer Logik und Schaltungsimplementationen.

Teil II

Lösungen mit asynchronen Kontrollpfaden

3 Micropipelines

Die von Ivan E. Sutherland [1] beschriebenen Micropiplines haben die herkömmlichen *instruction pipelines* von Microprozessoren zum Vorbild. Wie bei diesen werden Operationen parallel in den einzelnen Pipelinestufen ausgeführt und die Ergebnisse dann an die nächste Stufe weitergeben. Im Unterschied zu herkömmlichen Pipelines werden in Mircopipelines aber viel kleinere logische Einheiten verarbeitet. Der viel wesentlichere Unterschied ist jedoch, daß Micropipelinesstufen ihren nachfolgenden Stufen ihre Ergebnisse erst dann weitergeben, wenn diese signalisieren, daß sie dazu bereit sind.

Für die Implementierung verwendet Sutherland die im folgenden beschriebenen Konzepte.

3.1 Transition Signalling

Bei herkömmlicher Logik werden den beiden Signalzuständen *high* und *low* unterschiedliche Symbole zugeordnet und somit ein binärer Code übertragen. Beim *transition signalling* hat sowohl die steigende als auch die fallende Flanke eines Signals die gleiche Bedeutung und stellt ein sog. *event* dar. Der Nachteil ist, daß keine Daten kodiert werden können, nur *events*. Für diese besteht jedoch der Vorteil, daß wie beim NRZ (*non-return-to-zero*) Magnetaufzeichnungsverfahren das Zurückkehren in einen neutralen Zustand eingespart werden kann und somit Energie und Zeit gespart wird.

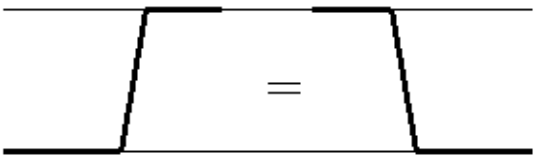

Abbildung 2: Transition Signalling

Die Folge dieser Symmetrie der Flanken ist, daß auch alle Schaltungen, die diese *event* verarbeiten, total symmetrisch im Bezug auf die Zustände *high* oder *low* sein müssen. Wie sich im Folgenden zeigen wird, ist besonders die CMOS Technologie aufgrund ihrer eigenen Symmetrie geeignet, *transition signalling* zu implementieren.

3.2 Event Logic

Transition signalling erfordert eigene Schaltungen zur Verarbeitungen von *events*. Die hierfür verwendete Logik wird *event logic* genannt. Das einfachste Gatter ist das OR-Gatter für *events*: es entspricht einem herkömmlichen (boolschen) XOR-Gatter. Wie sich leicht überprüfen läßt, gibt das XOR-Gatter genau dann ein *event* an seinem Ausgang aus, wenn sich einer seiner Eingänge ändert (also ein *event* auftritt).

Etwas komplizierter wird es beim AND-Gatter für *events*. Hierfür wird ein sog. Muller-C Gatter verwendet. Dieses ändert seinen Ausgang nur dann, wenn sich beide Eingänge geändert haben.

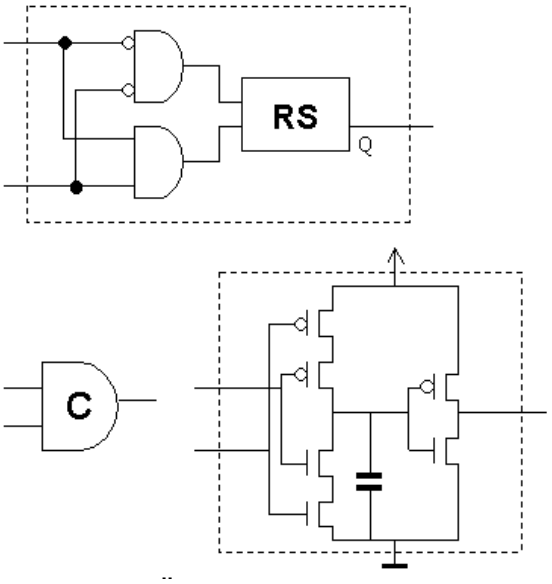

Abbildung 3: Ein Muller-C Element als Logikäquivalent, als Symbol und als Implementierung in dynamischer CMOS Technologie

Sein Verhalten läßt sich folgendermaßen beschreiben:

```
IF e1 == e2
THEN a = e1
ELSE a bleibt gleich
```

Obwohl der absolute Zustand eines *transition signal* keine Bedeutung hat, spielt der Zustand in Relation zu anderen Signalen sehr wohl eine Rolle. Daher existieren auch folgende Versionen von Muller-C Gattern:

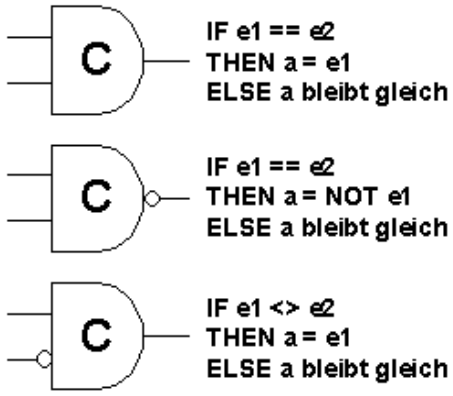

Abbildung 4: Verschiedene Varianten des Muller-C Elements die sich nur durch die Negationen an den Ein- und Ausgängen unterscheiden.

Es lassen sich weiters mit Hilfe von 10 bis 100 Gattern einige nützliche Bausteine für *event logic* entwerfen. Abbildung 5 zeigt einige im weiteren verwendete Module.

3.3 Event-Kontrollierte Register

Als Vorbereitung für die Einführung eines *event controlled storage element* definieren wir ein sog. *switch*-Gatter (Abbildung 6).

Mit Hilfe dieses „Schalters“ ist es nun einfacher, das folgende event-gesteuerte Speicherelement zu verstehen. Abbildung 7 zeigt zwei mögliche Implementierungen. Ein solches Speicherelement hat neben dem Datenein- und ausgang zwei Kontrolleitungen. Die Signale *capture* und *pass* führen in das Element hinein und werden nach der erfolgten Verarbeitung unverändert an den Leitungen *capture done* bzw. *pass done* wieder hinausgeführt.

Jedes Speicherelement besteht aus zwei Latches, die Abwechelnd aktiv sind. Im Ausgangszustand (beide control signals sind auf low) schaltet das Element seinen Eingang auf den Ausgang unverändert durch. In diesem Zustand verhält sich das Speicherelement (bis auf kleine Verzögerungen) wie eine di-

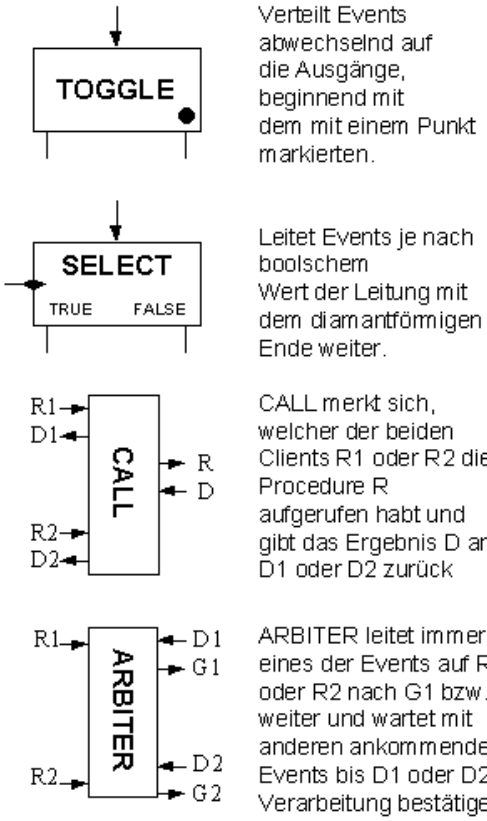

Abbildung 5: Nützliche Bausteine zur Verarbeitung von *event logic* .

rekte Leitung. Nach einem Signal auf Capture wird das aktive Latch vom Eingang getrennt und hält somit dieses Signal und liefert es an den Ausgang. Nach einem *pass* Signal wird das andere Latch aktiv und das Element ist (zwar spiegelverkehrt) wieder im Ausgangszustand. Auch hier ist festzuhalten, daß sich die Schaltung völlig symmetrisch im Bezug auf die absoluten Zustände der *event logic* verhält, nur die Relation spielt eine Rolle.

3.4 Two-phase bundled data convention

Die Grundlage für die Synchronisation zwischen den Pipelinestufen ist das von [7] beschriebene Protokoll, daß in [1] als *two-phase bundled data convention* bezeichnet wird.

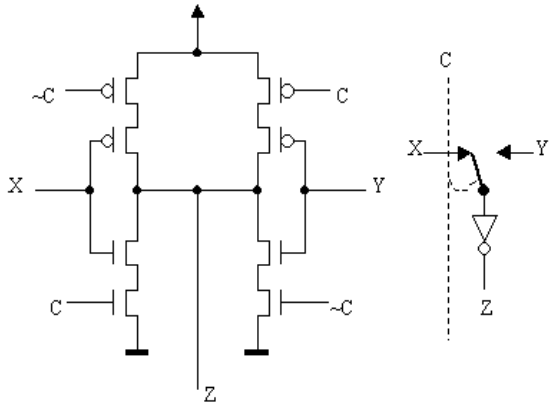

Abbildung 6: Schaltung für das Switch-Symbol. Ist C auf low, ist der Ausgang Z das Komplement von Y, ansonsten das Komplement von X. Das Symbol entspricht dabei dem Zustand bei C = low.

Die Übertragung findet zwischen einem Sender und einem Empfänger statt die durch eine beliebige Anzahl von Datenleitung und einer *request* und einer *acknowledge* Leitung verbunden sind. Abbildung 8 zeigt den prinzipiellen Aufbau.

Ein Übertragungszyklus schaut folgendermaßen aus:

1. Der Sender legt die zu übertragenden Daten auf die Datenleitungen.
2. Anschließend erzeugt er ein *event* auf der *request* Leitung.
3. Der Empfänger liest von den Datenleitungen und sendet nach dem Beenden des Lesevorgangs ein *event* auf der *acknowledge* Leitung.

Der Sender darf die Daten auf der Datenleitung nur in seiner aktiven Phase, also zwischen dem *acknowledge* des Empfängers und dem eigenen *request* ändern und muß diese sonst konstant halten.

3.5 Eine einfache Micropipeline

Mit Hilfe der Grundlagen aus dem vorigen Kapitel und in Anlehnung an die *two-phase data convention* soll nun eine Micropipeline konstruiert werden. Der allgemeine Aufbau schaut folgendermaßen aus:

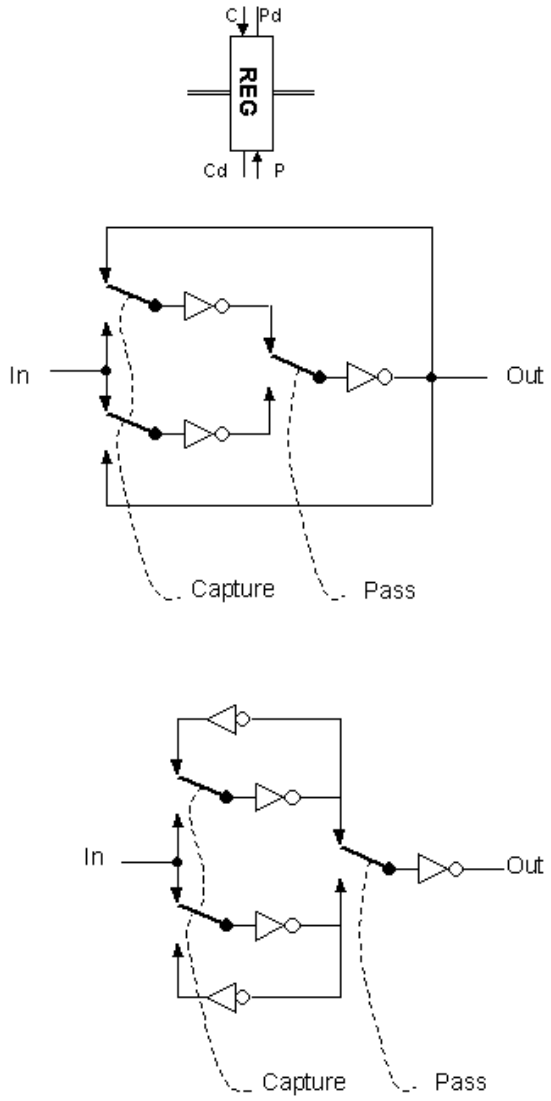

Abbildung 7: Symbol und zwei verschiedene Implementierungen für ein Event-gesteuertes Speicherelement. Die obere Variante benötigt weniger Gatter, ist aber etwas langsamer, da das Feedback durch zwei anstelle einem Schalter muß.

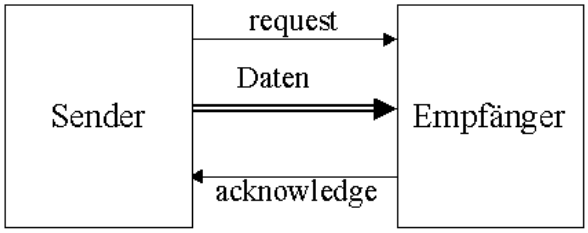

Abbildung 8: Ein bundled data Interface

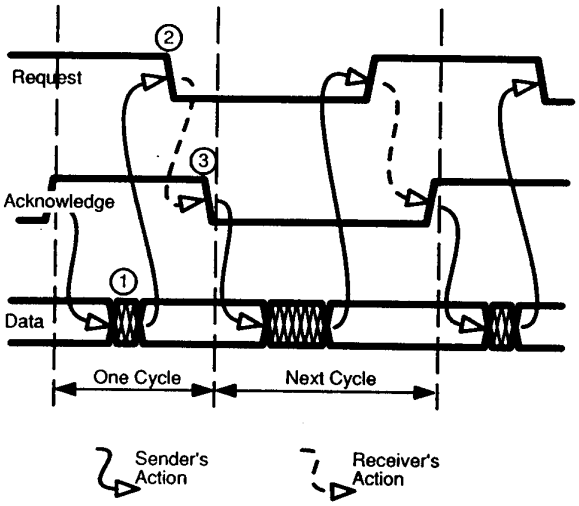

Abbildung 9: the two-phase bundled data convention

Die dünnen Linen stellen Kontrollpfade dar. Sie sind mittels *event logic* realisiert. Die Datenpfade (dicke Linien) verbinden wie bei einer Prozessorpipeline die Register zwischen den Stufen und die eigentliche Logik. Angenommen die Pipeline sei (z.B. nach einem *reset*) leer und alle Kontrollpfade auf *low*. Dann verhält sich die Pipeline als hätte sie keine Register - die Daten werden ohne Verzögerung durch die Logik geschleust. Kommt nun ein Signal auf R(in) - das heißt, es liegen neue gültige Daten auf D(in) - schaltet das erste Muller-C-Gate und das erste Register *captures* den Wert auf D(in). Anschließend wird auf A(in) die Übernahme quittiert und (über eine mit einem entsprechenden Delay versehene Leitung) ein Request R(1) an die folgende Stufe gesendet. Der übernommene Wert

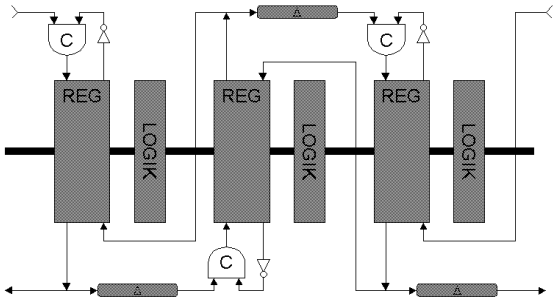

Abbildung 10: Schematischer Aufbau einer Micropipeline. Die hier gewählte Anordnung kann auch als IC-Layout verwendet werden.

läuft nun durch die Logik. Hierbei ist wichtig, daß die Laufzeit durch die Logik kleiner ist, als die Laufzeit durch das Delay-Element. Denn erreicht das Signal R(1) das Muller-C-Gate schaltet dieses und friert den von der Logik kommenden Wert D(1) ein, u.s.w.

Jede einzelne Stufe der Pipeline und auch die gesamte Pipeline verhält sich an ihren Ausgängen nach der *two-phase-bundled data convertion*. Somit braucht man lediglich mehrere Funktionseinheiten einfach aneinander zu hängen und die *acknowlede* und *request* Leitungen richtig zu verbinden - wenn die Teile funktionieren funktioniert auch das Ganze.

Als konkretes Beispiel soll nun ein (1 von 8)-Bit-Decoder gezeigt werden. Abbildung 11 stellt eine zweistufige Implementierung dar, die Schaltung von Abbildung 12 verwendet vier Pipelinestufen.

Die Kriterien für die Anzahl der Pipelinestufen sind ähnlich denen bei herkömmlichen Prozessorpipelines: Viele Stufen erlauben einen höheren *throughput* bei hohem Bauteilaufwand, weniger Stufen vermindern u.U. die *latency*, da weniger Register durchlaufen werden müssen.

3.6 Komplexere Schaltungen mit *event logic*

Neben einer einfachen Pipeline mit einem Dateneingang und einem Datenausgang bestehen weitere Möglichkeiten mittels *event logic* asynchrone Schaltungen zu entwerfen. Zum ersten ist es mög-

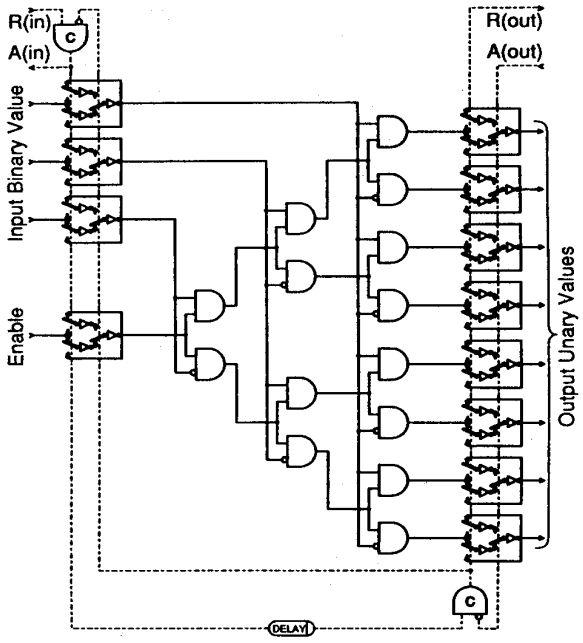

Abbildung 11: Decoder mit zwei Pipelinestufen

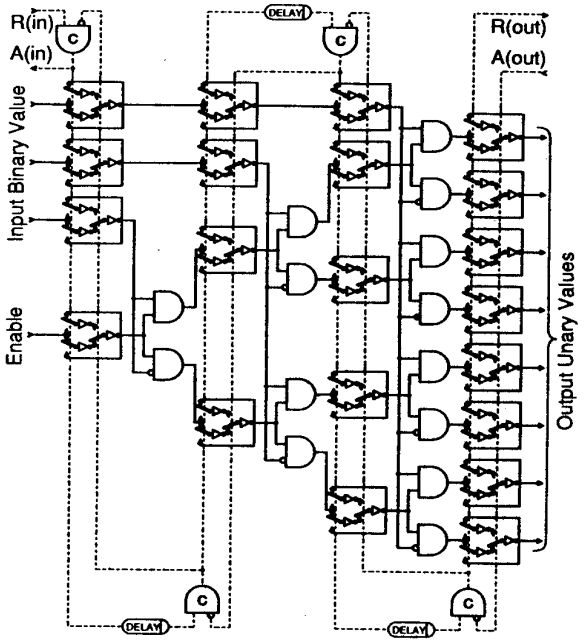

Abbildung 12: Decoder mit vier Pipelinestufen

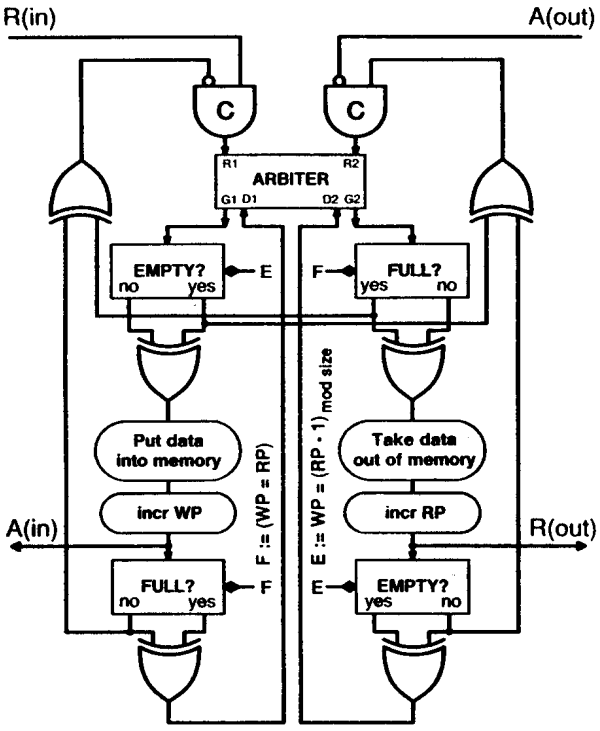

Abbildung 13: Kontrollogik für einen FIFO-Ringpuffer. Alle Leitungen bis auf die zu testenden Werte befördern *events*. Die Datenpfade sind nicht dargestellt.

lich, daß mehrere Pipelines in eine münden und umgekehrt. Dies ist vor allem mit Hilfe der Bauteile aus Abbildung 5 leicht möglich. Es sind aber auch noch viel komplexere Kontrollstrukturen mittels *event logic* möglich. Dabei läßt sich der Aufbau praktischerweise meist aus dem Kontrollflußgraphen direkt ableiten. Als ein Beispiel hierfür soll eine Ringpufferverwaltung gezeigt werden (Abbildung 13). Für weitere Beispiele wird auf [1] verwiesen.

4 Weitere Verfahren

4.1 Bounded delay model

Das *bounded delay model*, auch als *delay insensitve (DI) modell* bekannt, nimmt für gewisse Logikmodule konstante Obergrenzen für die Durchlaufzeit an. Aufgrund dieser Abschätzung wird dann eine Schaltung entworfen, die nach einem *request* diese Zeitdauer abwartet und dann das *acknowledge* Signal sendet. Diese Laufzeit muß für den worst-case ausgelegt sein, überschreiten Parameter wie hohe Temperatur oder niedrige Spannung diesen, funktioniert die Schaltung u.U. nicht mehr. [3]

4.2 Unbounded delay model

Beim *unbounded delay model*, das auch *scalable delay insensitive (SDI) model* genannt wird, wird keine Obergrenze für die Ausführungszeit einer kleinen logischen Einheit festgelegt, sondern eine Grenze für die Abweichung des Verhältnisses der Ausführungszeit zweier Einheiten. [12]

In gewisser Hinsicht lassen sich Ähnlichkeiten zwischen dem SDI-Modell und den Mikropipelines finden, denn auch bei letzeren wird die Laufzeit einer Schaltung durch eine andere Schaltung (*delay*) approximiert und angenommen, daß die Laufzeit durch das Logikgatter die des Delay-Elements nicht übersteigt.

4.3 Current Sensing Completion Detection (CSCD)

Der CSCD-Ansatz versucht über die Stromaufnahme von Gattern festzustellen, ob diese eine Operation bereits beendet haben. Bei der CMOS-Technologie ist die Stromaufnahme beim Schalten größer als im stabilen Zustand. Wird diese Stromaufnahme gemessen kann eine Aussage über die Beendigung einer Berechnung gemacht werden. Meist wird diese Methode mit einem Verzögerungselement für den *worst-case* Fall nach dem *bounded delay model* (s.Kapitel 4.1) kombiniert, um für den Fall, daß die Beendigung nicht erkannt wurde, weitergearbeitet werden kann. [3]

5 Implementierungen

Mir sind bis zum jetzigen Zeitpunkt zwei fertige Mikroprozessoren als Implementierungen von asynchroner Logik mit Hilfe von Micropipelines bekannt. Es sind dies der von der University of Manchester entwickelte AMULET der nun bereits als dritte Version (AMULET3) vorliegt und der von der University of Tokyo entworfene TITAC-2.

5.1 AMULET

Die erste Version des AMULET - der AMULET1 - wurde im Jahre 1994 im Rahmen einer Dissertation entwickelt. Es war eine asynchrone Implementierung einer ARM 32-bit RISC Architektur. Dieser Chip diente lediglich dem Nachweis, daß die entwickelten Tools und Konzepte auch auf einen ganzen Prozessor aufweitbar sind und schöpfte nicht alle Möglichkeiten eines asynchronen Designs aus [8]. Abbildung 14 zeigt dem interen Aufbau des Prozessors.

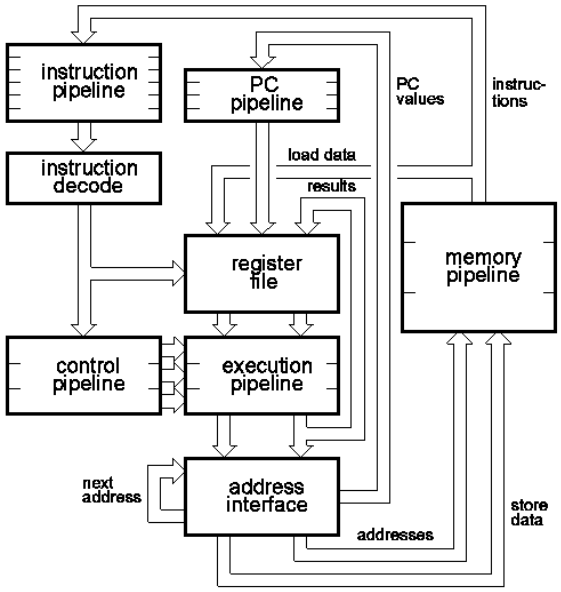

Abbildung 14: Der interen Aufbau des AMULET1.

Aus den Erfahrungen mit dem AMULET wurden folgende Erkenntnisse gewonnen:

- Der durchgehende Einsatz von Micropipelines hat sich als vernünftig im Bezug auf Design und Optimierung erwiesen.
- Was im Prozessor gut funktionierte erwies sich auf dem Motherboard als eher problematisch: Das Debuggen des fertigen Prozessors mit den beiden two-phase Signalen war sehr schwierig und Zeitaufwendig.
- Die verwendete CMOS-Technologie erwies sich für die *event logic* als etwas ineffizient: Alle Register wandelten das two-phase Signal intern in ein four-phase Signal um. Der Grund hierfür ist wohl, daß CMOS grundsätzlich eine levelsensitive Technologie ist und Events nicht direkt verarbeitet werden können.
- Lange Pipelines zu konstruieren erwies sich als zu leicht, was sich auf die Performance auswirkte.

Unter Berücksichtigung dieser Punkte wurde der AMULET2 entworfen: es wurde die *two-phase conversion* zugunsten einer *four-phase conversion* aufgegeben, konsequent auf die Pipelinelänge geachtet und dem Systeminterface an den Ausgängen mehr Beachtung geschenkt. Weites wurden einige Veränderungen an der Architektur zugunsten der Performance und des Stromverbrauchs vorgenommen. Genaueres hierzu findet sich in [8].

Der AMULET2e ist ein Embedded Controller mit einem AMULET2 core. Auch er funktioniert vollständig mittels Micropipelines. Seine Eckdaten sind:

- Kompatibel zum ARM v4G Instruktionssatz
- 4 kB Speicher On-Chip, als RAM oder Cache konfigurierbar
- Direktes Inteface für statischen und dynamischen Speicher (nicht-asnychron!)
- 128 pin PQFP (plastic quad flat package) oder 144 TQFP
- 0.5 μCMOS, 3 layer metal process
- 3.3V Versorgungsspannung

Der Vergleich mit zwei (den gleichen Instruktionssatz verwendenden) ARM-Chips zeigt, daß das asynchrone Design im Bezug auf Performance durchaus mit synchronen Entwurfen konkurrieren kann:

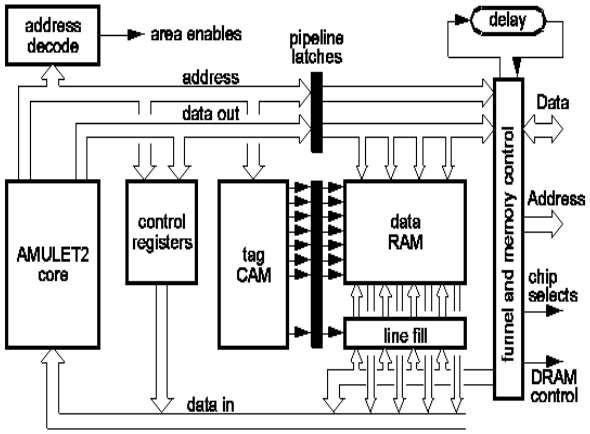

Abbildung 15: Aufbau des AMULET2e embedded controller

	ARM710	AMULET2e	ARM810
Prozess	0.6 μm 2LM	0.5 μm 3LM	0.5 μm 3LM
Fläche mm^2	32	41	76
Transistoren	570.295	454.000	836.022
Cache	8K 4-way	4K 64-way	8K 64-way
MIPS	23	40	86
Bedingungen	35 MHz	20°C	72 MHz
mW	120	150	500
MIPS/W	192	250	172

Der neueste Chip in dieser Familie ist nun der AMULET3. Er hat ein überarbeitetes Design und viele neue Features ohne die ein moderner Prozessor nicht mehr auskommt (wie z.B einen *branch target buffer* und einen *reorder buffer*) und soll im Unterschied zu seinen Vorgängern ein komerzielles Produkt sein. Er unterstützt die neueste Version des ARM Instruktionssatzes. Gefertigt wird er in einem 0.35 μ3LM Prozeß und soll auf über 100 MIPS (Dhrystone 2.1) kommen. [10]

5.2 TITAC-2

TITAC-2 [12] ist ein asynchroner 32-Bit Mikroprozessor der von der University of Tokyo entwickelt wurde. Er hat seine Architektur vom MIPS R2000, ist allerdings nicht Codekompatibel, da er einen etwas anderen Instruktionssatz hat. Er verwendet er ein Unbounded Delay model. Seine Eckdaten sind:

- 3 metall layer
- 0.5 μ CMOS
- 496.367 Transistoren
- 12.5 mm x 12.5 mm

Der Chip funktionierte korrekt mit Versorgungsspannungen von 1,5V bis 6,0V und Temperaturen von 85°C bis zu der von flüssigem Stickstoff. Bei Raumtemperatur und 3.3V Corespannung erreichte der Dhrystone Benchmark 53 VAX MIPS bei einem Stromverbrauch von 2W.

Teil III
Lösungen mit asynchronen Datenpfaden

6 Double rail Codierung

Double rail encoding ist das älteste Konzept dieser Gruppe. Hierbei werden für jede logische Datenleitung zwei Leitungen verwendet.

6.1 Konzept

Beim *double rail encoding* wird neben den Zuständen 0 und 1 - die als *defined* bezeichnet werden sollen - ein dritter logischer Zustand - *undefined* oder *intermediate* - eingeführt. Codiert wird diese dreiwertige Logik mittels zwei binären Leitungen. Dabei wird folgende Abbildung verwendet:

$$U \mapsto 00$$
$$0 \mapsto 10$$
$$1 \mapsto 01$$

Der Wert 11 ist nicht zulässig und dürfte in einer korrekten Schaltung auch nicht auftreten. Entscheidend ist, daß bei einem Übergang von 0 nach 1 (wenn man eine genügend feine Auflösung im Zeitbereich betrachtet) immer der Zustand U passiert wird. Eine Schaltung soll an ihren Ausgängen erst *defined* werden, wenn alle ihre Eingänge *defined* sind. Ebenso sollen ihre Ausgänge erst *undefined* werden, wenn alle Eingänge *undefined* sind. Somit ergibt sich folgender Ablauf:

1. Alle Eingänge sind *undefined*
2. Alle Ausgänge werden *undefined*
3. Einige Eingänge werden *defined*
4. Alle Ausgänge bleiben *undefined*
5. Alle Eingänge sind jetzt *defined*
6. Alle Ausgänge werden *defined*
7. Einige Eingänge werden *undefined*
8. Alle Ausgänge bleiben *defined*
9. *weiter mit 1.*

Wenn alle Teilschaltungen sich nach diesem Muster verhalten, synchronisieren sie sich dadurch selbst und bilden ein *self timed system.*

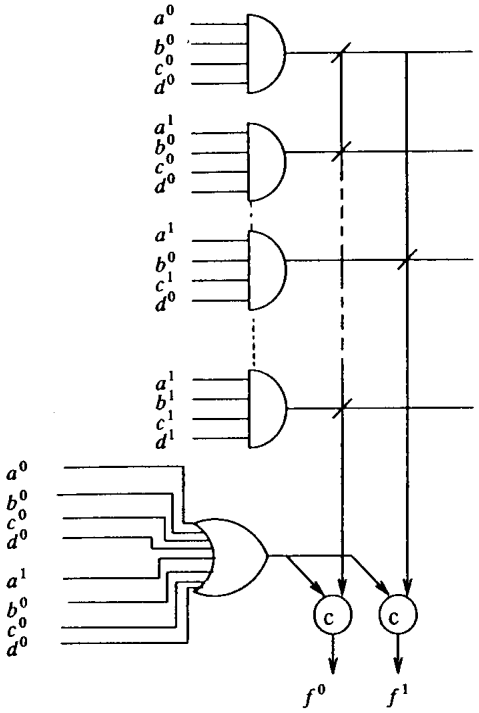

Abbildung 16: Beispiel für eine Implementierung nach Seitz.

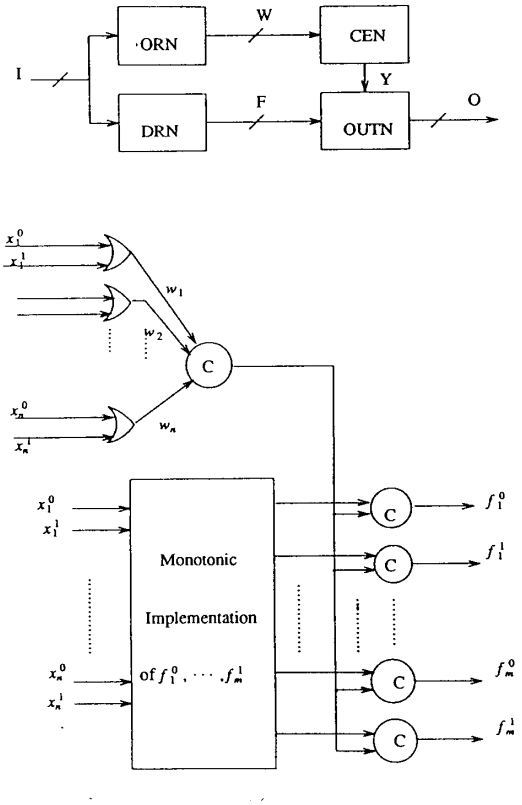

Abbildung 17: Blockbild für die verschiedenden Subnets.

6.2 Implementierung nach Seitz

Seitz beschreibt eine einfache, aber u.U. nicht sehr effektive Implementierungsmöglchkeit für *double rail encoding* [7]. Dabei wird ähnlich einem PLA die Funktion über das Bilden aller Kombinationen von Eingängen und Verknüpfung mit den Ausgangsleitungen gebildet. Abbildung 16 zeigt ein Beispiel. Mittels der OR und des Muller-C-Gates wird sichergestellt, daß der Ausgang *defined* bleibt, solange nicht alle Eingänge *undefined* sind.

6.3 Effizientere Implementierung nach [2]

Bei dieser Implementierungsvariante wird die zu entwerfende Schaltung in vier Blöcke untergliedert (Abbildung 17). Diese *subnets* genannten Teile werden nach folgendem Schema entworfen.

Angenommen es existieren n double-rail Eingänge und m double-rail Ausgänge:

Subnet ORN besteht aus n OR-Gatter mit 2 Eingängen und hat die Aufgabe festzustellen, ob auf den Daten ein definierter Wert liegt.

Subnet CEN besteht aus einem einzelnen Muller-C-Element mit n Eingängen. Der Ausgang dieses Subnets wird 1, wenn alle Eingänge *defined* sind und 0, wenn alle Eingänge *undefined* sind.

Subnet DRN In diesem Subnet wird die eigentliche Funktion aus den Eingängen in minimale Darstellung gebildet. Diese ist trotz mehrerer Eingänge meist nicht viel komplexer als in konventioneller boolscher Logik. Sei $f(x_1, ...x_n)$ die zu darstellende Funktion. Dann wird die minimale Darstellung für f und f' auf bekanntem Wege aus $x_1...x_n$ und $x'_1...x'_n$ gebildet und die Eingänge entsprechend auf $x^1_1...x^1_n$

bzw. $x_1^0 ... x_n^0$ umbenannt. Wenn man die Ausgänge f und f' auf F^1 und F^0 umbenennt entsprechen diese genau der Funktion für double rail kodierte Daten.

Subnet OUTN besteht aus $2m$ Muller-C-Gates, die die berechnete Funktion nur bei definierten Eingängen durchlassen.

Diese Variante kommt in einem kontreten (einfachen) Beispiel in [2] auf deutlich weniger Gatter als die von Seitz oder anderern Autoren beschriebenen Varianten. Ob dies generell gilt, müßte überprüft werden. Für einfache Funktionen, ist diese Variante sicher effektiver.

7 Tree rail Codierung

Will man asynchrone Logik ohne Elemente mit internem Speicher oder Feedback (wie z.B. das Muller-C-Gate) sondern ganz normale boolsche Gatter, die ihr timing selbst bestimmen, braucht man eine dreiwertige Logik [9]. Dabei muß folgende Codierung (oder Permutationen davon) verwendet werden:

$$0 \mapsto 011$$
$$1 \mapsto 110$$

Unter Verwendung der Gatter aus Abbildung 18 kann ganz normale boolsche Logik weiter verwendet werden. Da diese Gatter überdimensional viele Leitungen und Gatter brauchen ist dieser Ansatz wohl eher theoretisch.

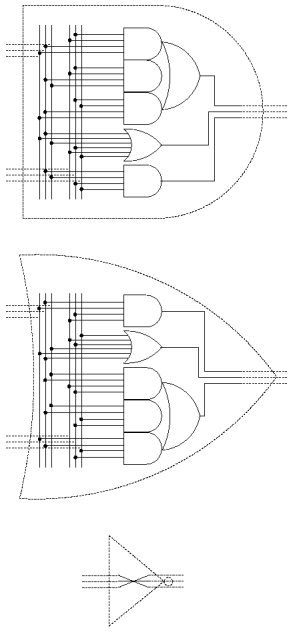

Abbildung 18: AND, OR und NOT in *three rail encoding*.

8 NULL-Convention-Logic

Die NULL-Convention-Logic [5] hat ihre Wurzeln im *invokation model of process expression*, einem konzeptionellen Modell zur Beschreibung von Prozessen. Mit diesem Modell werden von chemischen und phyikalischen Vorgängen bis hin zu komplexen natürlichen oder künstlichen Prozessen modelliert. Computer stellen für dieses Modell also nur eine von mehreren Anwendungsgebieten dar.

Auch die NULL-Convention-Logic erweitert die Boolsche Logik um ein bzw. zwei Symbole. Im Gegensatz zum *double rail encoding* wird das Konzept auf kleinster logischer Ebene angewandt. Dabei werden neue Elementargatter - die sog. *treshold gates* definiert und mit diesen größere Bauteile konstruiert.

8.1 Konzept

Bei der boolsche Logik sind die beiden Zustände *true* (T) und *false* (F) *mutual exclusive*. Egal wie die Zustände interpretiert werden, repräsentieren diese nur Daten. Es existeren keine Symbole für Nicht-Daten oder Kontrolle im Wertebereich der boole'schen Logik. Nach dem *invokation model of process expression* muß ein neuer Wert hinzugefügt werden, welcher in der NULL-Convention-Logic mit NULL (N) bezeichnet wird. Die Werte T und F werden *data values*, der Wert N *non-data value* genannt. Die sich dadurch ergebenden Wertetabellen für die boole'schen Operatoren AND, OR und NOT sehen so aus:

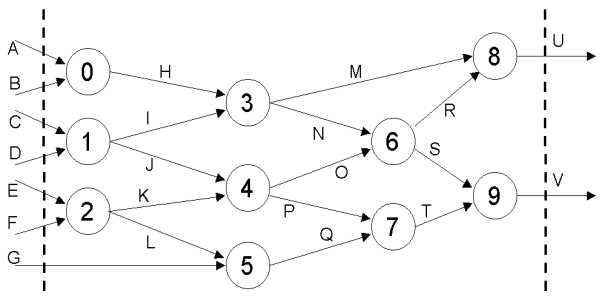

Abbildung 19: Symbolbild für mehrere Gatter die Abhängigkeiten untereinander haben.

$\cap$	T	F	N
T	T	F	N
F	F	F	N
N	N	N	N

$\cup$	T	F	N
T	T	T	N
F	T	F	N
N	N	N	N

$\neg$	
T	F
F	T
N	N

Die Bedingungen sind somit die gleichen wie beim *double rail encoding* im Bezug auf den *intermediate* Wert: Erst wenn sich alle Eingänge geändert haben, ändert sich auch der Ausgang.

Ausgehend von einem Netz, in dem alle Leitungen auf NULL sind, breiten sich die gültigen Inputwerte wie eine Welle bis zu den Ausgängen aus. Durch einfaches Überwachen der Ausgänge läßt sich leicht feststellen wann die Berechnung abgeschlossen ist, nämlich dann, wenn alle Ausgangsleitungen von NULL verschieden sind.

Da, um eine neue Berechnung beginnen zu können alle Leitungen wieder auf NULL sein müssen, muß nach Ende der Berechnung eine zweite Welle das System wieder komplett auf NULL setzen.

Dies kann die oben definierte Algebra nicht erfüllen. Zur Lösung dieses Problems gibt es zwei Möglichkeiten:

1. Es wird ein vierter Wert (*intermediate* eingeführt. Dadurch ergeben sich folgende Wertetabellen:

$\cap$	T	F	I	N
T	T	F	I	I
F	N	F	I	I
I	I	I	I	I
N	I	I	I	N

$\cup$	T	F	I	N
T	T	T	I	I
F	T	F	I	I
I	I	I	I	I
N	I	I	I	N

$\neg$	T	F	I	N
	F	T	I	N

Damit ist es möglich beide Arten von Wellen durch die Schaltung laufen zu lassen. Der Beobachter am Ende wartet auf DATA oder NULL und ignoriert alle *intermediate*-Werte.

2. Jedes Gatter hat eine Feedback, d.h. sein Ausgang wird als zusätzlicher Eingang rückgeführt. Das erlaubt dem Gatter seinen Zustand zu merken und nur dann von NULL zu DATA zu wechseln, wenn alle Eingänge DATA sind, aber bei DATA zu bleiben wenn dann ein Eingang NULL wird. Andererseits wechselt das Gatter von DATA zu NULL wenn alle Eingänge NULL sind und bleibt dabei, auch wenn sich ein Eingang zu DATA ändert.

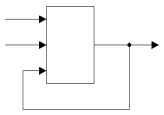

Abbildung 20: Gatter mit Rückführung.

Die erste Lösung ist total *delay insensitive*, die zweite nicht ganz. Sollte ein Feedback langsamer sein, als eine ganze *wavefront* würde dieser Ansatz nicht funktionieren. Da dies aber im Normalfall nicht auftritt - die Laufzeit durch das gesamte Gatter ist um vieles länger als ein einfaches Feedback - ist diese Lösung *effectively delay insensitive*.

8.2 Implementierung

Da weder eine drei- noch vierwertige Logik existiert, muß diese auf zweiwertige Logik abgebildet werden. Da der Zustand *low* dem Wert NULL zugeordnet wird, bleibt nur ein Zustand *high* für DATA übrig. Mit mehreren Leitungen lassen sich aber auch mehr verschiedene Werte kodieren. Da in so einer Gruppe nur eine Leitung auf DATA sein darf, nennt man so eine Gruppe auch *mutually exclusive assertation group*. Die Größe einer solchen Gruppe kann beliebig groß sein, so könnte man z.B eine Dezimalziffer mit 10 Leitungen kodieren. Hat die Gruppe nur zwei Leitungen (wie es bei der

Feedback-Lösung der Fall ist) ist das Ergebnis das Gleiche wie beim *double rail encoding*:

$$N \mapsto 00$$
$$F \mapsto 10$$
$$T \mapsto 01$$

Der Unterschied besteht darin, daß das *double rail encoding* nur ein Protokoll zwischen den Schaltungen ist, bei der NULL-Convention-Logic ist diese Kodierung Teil der Logik selbst.

Da bei der NCL nur ein Datenwert existiert, ist die einzige Funktion, die ein Gatter haben kann, zu zählen, wieviele Eingänge auf Data sind. Ein Gatter, das diese Funktion erfüllt wird *dicrete threshold gate* genannt.

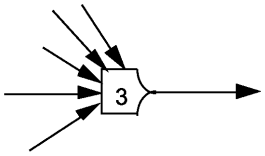

Abbildung 21: Ein 5 input threshold 3 gate.

Ein solches *threshold n gate* ist genau dann DATA wenn mehr als n Eingänge auf DATA sind, ansonsten NULL. Wie sich leicht nachprüfen läßt sind das AND oder OR Gatter Spezialfälle dieses Gatters (2 input threshold 2 gate bzw. 2 input threshold 1 gate).

Als Beispiel zeigt Abbildung 22 nun ein Halbaddierer sowohl in konventioneller als auch in NULL-Convention-Logic.

Mit diesem Gatter läßt sich leicht eine Schaltung bauen, die zwar DATA-Wellen richtig verarbeiten kann, aber wiederum keine NULL-Wellen. Die Anwendung der Lösungsansätze aus Kapitel 8.1 führt zu folgenden Möglichkeiten:

1. Durch die Einführung des vierten Wertes bei der *intermediate*-Lösung wird folgende Kodierung vorgenommen:

$$enc\_DATA \mapsto DATA, DATA$$
$$enc\_INTERMEDIATE \mapsto DATA, NULL$$

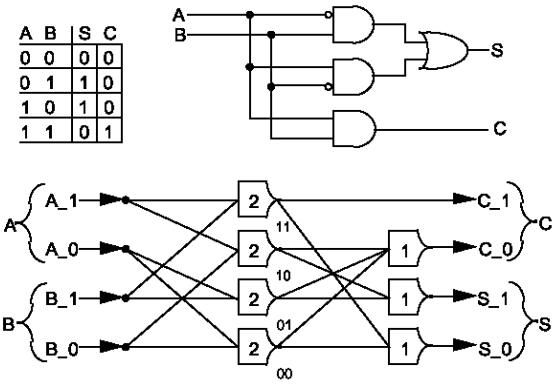

Abbildung 22: Ein Halbaddierer mit Wahrheitstablle, in boolscher und NULL-Convention-Logic.

$$enc\_INTERMEDIATE \mapsto NULL, DATA$$
$$enc\_NULL \mapsto NULL, NULL$$

Diese Kodierung ist eine Ebene über den *mutually exclusive assertation groups* und somit braucht jeder Wert vier Leitungen. (z.B.: $0 \mapsto (enc\_DATA, enc\_NULL) \mapsto ((DATA, DATA), (NULL, NULL))$). Auch die speziellen *threshold gates* für diese Implementierung braucht viele Leitungen und Gatter [5] und daher ist diese Lösung eher uninteressant. Sie soll daher nicht weiter behandelt werden.

2. Jedes *threshold gate* mit dem threshold Wert n wird mit einem Feedback vom Gewicht n-1 versehen. Abbildung 23 zeigt ein Beispiel und das Symbol für ein *threshold gate* mit Feedback.

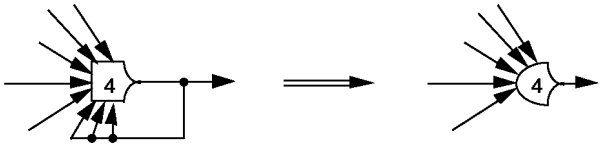

Abbildung 23: Ein threshold gate mit feedback.

Dieses Gate wird erst DATA wenn n Eingänge DATA sind aber auch erst NULL wenn alle

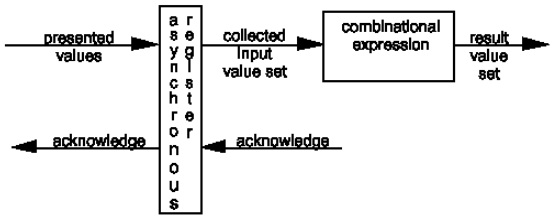

Abbildung 24: Blockbild eines asynchronen Registers.

Eingänge NULL sind. Das Muller-C Gate ist übrigens ein Spezialfall dieses Gates.

8.3 Ein asynchrones Register mit NCL

Ein asynchrones Register muß folgende Aufgaben erfüllen:

- Die Fertigstellung einer DATA/NULL Welle erkennen.
- Dem Vorgänger melden, daß die DATA/NULL Welle fertig ist.
- Wenn der Nachfolger meldet, daß eine DATA/NULL Welle angekommen ist, die nächste NULL/DATA Welle starten.

Das hierzu gehörende Blockbild ist in Abbildung 24 und eine einfache Implementierung in Abbildung 25 zu sehen.

Angenommen es wurde gerade eine NULL-Welle durch die folgende Schaltung geschickt. Meldet nun das nächste Register die Vervollständigung des Durchlaufs wird die *acknowledge* Leitung von Nachfolger auf DATA gesetzt (durch die Negation) und daher können anliegende DATA-Werte das Register passieren. Erst wenn alle vier Gruppen auf DATA sind schaltet das *threshold 4 gate* und die Leitung zum Vorgängerregister wird auf NULL gesetzt. Sind nun alle Werte durch die Logik gewandert und am nächsten Register angekommen und ist dieses bereit, liefert es eine NULL auf der *acknowledge* Leitung und dieses Register ist nun bereit, die NULL Welle passieren zu lassen.

Nach diesem Schema können nun wieder Register und Logik zu einer Pipeline kombiniert werden.

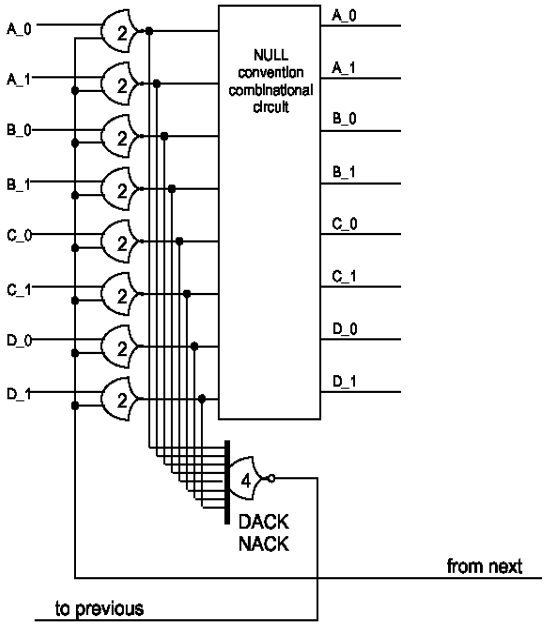

Abbildung 25: Ein asynchrones NCL Register.

8.4 Realisierungen von theshold gates

Die Effizienz der NCL steht und fällt mit dem Aufbau der *threshold gates* mit Feedback. In [15] finden sich Anleitungen für die Implementierung in statischer, semi-statischer und dynamischer CMOS-Technologie.

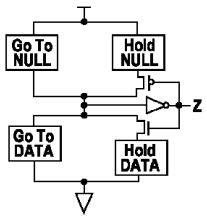

Abbildung 26: Konzeptioneller Aufbau eines *threshold gates* mit Feedback in statischer CMOS-Technologie. Die einzelnen Blöcke enthalten die Bedingungen, bei denen das Gatter zu NULL bzw. DATA wird und deren Komplemente.

Teil IV
Schluss

9 Schlußbetrachtungen

9.1 Zusammenfassung

In der Literatur finden sich mehr oder weniger verschiedene Ansätze für Schaltungen ohne Takt. Während es für die in Abschnitt II beschriebenen Konzepte auch nur einen praktisch relevanten Ansatz - die Micropipelines von Ivan E. Sutherland - gibt, existieren für die Konzepte in Abschnitt III viele verschiedene Ansätze, die sich untereinander allerdings nicht sehr unterscheiden.

Micropipelines arbeiten ähnlich wie herkömmliche Instruktionpipelines, nur werden sie nicht durch einen Takt gesteuert, sondern teilen sich selbst untereinander mit wann sie bereit sind, Daten zu lesen bzw. wann das Lesen abgeschlossen ist. Für diese Technologie existieren schon fertige Prozessoren, allerdings verwenden die neueren Versionen nicht mehr die *event logic*, wie von Sutherland beschrieben.

Konzepte, bei denen die Kontrollinformation in die Daten integriert werden, führen ein oder mehrere zusätzliche logische Symbole ein und kodieren diese auf mehreren Leitungen. Bei der NULL Convention Logic wird ein neues Elementargatter - das *discrete threshold gate* - eingeführt. Daher müssen Schaltungen (z.B. Addierer) komplett neu entworfen werden.

Wird nur ein *double rail encoding* durchgeführt, wie z.B. von Seitz und David vorgeschlagen, kann mit herkömmlichen Gattern gearbeitet werden, die Kontrollinformation wird erst am Schluß über ein Muller-C-Gate hinzugefügt.

9.2 Vergleich zwischen verschiedenen asynchronen Techniken

Mciropipelines stellen ein lange bekanntes und ausgereiftes Konzept dar. Auch daß die einzigen marktreifen Prozessoren mit dieser Technologie arbeiten spricht für sich. Aus meiner Sicht hat diese Technologie folgende Vor- und Nachteile:

+ Die Datenleitungen sind in konventioneller boolscher Logik ausgeführt. Dadurch können bekannte Schaltungen leichter portiert werden. Auch neue Schaltungen können mit vorhandenen Tools und Bibliotheken erzeugt werden. Sollte wie im Falle des AMULET2e und AMU-

LET3 sogar auf die *event logic* verzichtet werden, ändert sich im Vergleich zu einem herkömmlichen Design nur wenig.

+ Aus diesem Grund bleibt auch der Gatteraufwand in der gleichen Größenordnung wie bei äquivalenten synchronen Schaltungen. Lediglich die neuen Register könnten u.U. etwas mehr Gatter benötigen.

+ Der Entwurf der Kontrollogik läßt sich nach [1] oft direkt aus dem Kontrollflußgraphen ableiten. Dies erleichtert die Entwurfsphase sicherlich enorm. Generell kommt mir die Event-gesteuerte Kontrollogik intuitiver vor.

+ Jeder moderne Prozessor arbeitet mit Pipelines. Micropipelines sind nur eine konsequente Fortsetzung dieses Konzepts und passen gut in bekannte Architekturen.

– Micropipelines brauchen ein Verzögerungselement, daß die Laufzeitzeit durch die Logikgatter abschätzt. Somit sind Micropipelines nicht unabhängig von Gatterlaufzeiten, nur wird angenommen, daß sich die Laufzeit des delay-Elements sich gleich verändert wie die der Logik (s. SDI-Modell, Kapitel 4.2).

– Micropipelines verleiten zu zu langen Pipelines.

NCL ist eine neuere Technik und somit vielleicht noch nicht so weit entwickelt. Dieses Konzept hat einen großen theoretischen Hintergrund, nicht nur in der Informatik. Seine Vor- und Nachteile sind:

+ NCL ist ein wirklich asynchroner Ansatz, die Kontrollinformation wird in die Logik selbst integriert. Ein weites Betätigungsfeld für Theoretiker und Mathematiker.

+ NCL braucht keine Verzögerungselemente.

+ Das Design erwies sich als leicht portierbar zwischen verschiedenen Halbleitertechnologien [6].

+ Auch NCL braucht nicht unbedingt mehr Elementargatter als konventionelle Logik, in manchen Fällen (wie z.B. bei einem Volladdierer) sogar deutlich weniger.

– Anders sieht es mit den Leitungen aus: es werden sicher doppelt so viele benötigt.

– Auf jede DATA Welle muß eine NULL Welle folgen. Das halbiert den Durchsatz durch die Schaltung.

Alle anderen Konzepte erwiesen sich als eher theoretischer Natur: für sie existiert nach meinem Wissen kein Silkon.

9.3 Resumee

Asynchrone Schaltungen sind nicht nur ein theoretisches Konzept sondern funktionieren auch in der Praxis recht gut. Performancegewinne darf man sich apriori aber nicht erwarten: meist benötigen die Schaltungen doch einige Gatter mehr und durch die aynchrone Implementierung werden die Signale auch nicht schneller. Allerdings brauchen asynchrone Schaltungen weniger Strom was gerade im Bereich des Mobile Computing und für Embedded Systems ein entscheidender Vorteil sein könnte.

Performancegewinne gibt es nicht durch drehen an der „Taktschraube“sondern nur durch eine günstige Architektur und vorallem eine niedere Temperatur. Vielleicht wird ja auch mal der Kühler das Herzstück eines schnellen Rechners...

Aus meiner Sicht bringt aber gerade im Bereich des Entwurfs asynchrone Logik einen großen Vorteil. So kann an einem bestehenden Layout eine Komponente (z.B. wenn sich ein Teil als Bottleneck erwiesen hat) einfach ausgetauscht werden, es muß sich weder über globale Taktverteilung Gedanken machen, noch ob diese neue Komponente auch in einem Taktzyklus fertig wird.

Im Marktsegment für Embedded Systems räume ich diesen Techniken eine gute Chance ein, ob diese Konzepte aber auch auf dem PC Markt in absehbarer Ziet Einzug finden, möchte ich schwer bezweifeln.

Literatur

[1] Ivan E. Sutherland, *„Micropiplines“*, in „Communications of the ACM“, Vol. 32, No. 6, June 1989, pp. 720-738.

[2] Illana David, Ran Ginosar und Michael Yoeli, *„An Efficient Implementation of Boolean Functions as Self-Timed Circuits“*, in „IEEE Transactions on Computers“, Vol. 41, No. 1, January 1992, pp. 2-10.

[3] Andreas Beul, *„Wie taktlos“*, in „c't magazin für computer technik“, Ausgabe 17/99, Verlag Heinz Heise

[4] Fred. U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, Ting-Pien Fang, *„Q-Modules: Internally Clocked Delay-Insensitive Modules“*, in „IEEE Transactions on Computers“, Vol. 37, No. 9, September 1988, pp. 1005-1018.

[5] Karl M. Fant, Scott A. Brant, *„NULL Convention Logik“*, Theseus Logic, Inc. , http://www.theseus.com

[6] Dr. Ching-Yi Wang, Dave Parker, Ryan Jorgenson and Karl Fant, *„Technology Independent Design Using NULL Convention Logic“*, Theseus Logic, Inc. , http://www.theseus.com

[7] C. L. Seitz, *„System Timing“*, in Mead, Carver A., „Introduction to VLSI Systems“, Addison-Wesley, 1980.

[8] S. B. Furber, J. D. Garside, u. a. *„AMULET2e: An Asynchronous Embedded Controller“*, Department of Computer Science, The University of Manchester, http://www.cs.man.ac.uk/amulet

[9] Märt Saarepear, Tomohiro Yoneda, *„A Self-Timed Implementation of Boolean Functions“*, Department of Computer Science, Tokyo Institute of Technology

[10] J. D. Garside, S. B. Furber and S-H Chung *„AMULET3 Revealed“*, Department of Computer Science, The University of Manchester

[11] *„Amulet2e datasheet“*, Department of Computer Science, The University of Manchester

[12] TITAC-2, A 32-bit Asynchronous Microprocessor Department of Computer Science, Tokyo Institute of Technology

[13] W.J.Bainbridge and S.B.Furber, *„Asynchronous Macrocell Interconnect Using MARBLE“*, Department of Computer Science, The University of Manchester

[14] B. Moszkowski, *„A temporal logic for multilevel reasoning about hardware“*, in „IEEE Computer Magazine“, pp. 10-19, Feb. 1985

[15] Gerald E. Sobelman, Karl Fant, *CMOS circuit design of threshold gates with hysteresis*, Theseus Logic, Inc. , http://www.theseus.com