

## **Ein Framework zum Entwurf Applikationsspezifischer RISC Prozessoren (ASPs)**

J. Reichardt, B. Schwarz

Fachbereich Elektrotechnik und Informatik der Hochschule für Angewandte Wissenschaften Hamburg,  
Berliner Tor 7, D-20099 Hamburg

[reichardt@etech.haw-hamburg.de](mailto:reichardt@etech.haw-hamburg.de), [schwarz@etech.haw-hamburg.de](mailto:schwarz@etech.haw-hamburg.de)

**Kurzbeschreibung:** Dieser Beitrag beschreibt Entwurfsrichtlinien zum strukturierten Entwurf applikationsspezifischer Prozessoren (ASPs). Ziel dieses Frameworks ist es, den Entwurfsprozess mit dem Ziel einer schnellen ASIC-Realisierung möglichst transparent zu gestalten. Exemplarisch wird die Implementierung zweier unterschiedlich komplexer Instruktionssätze vorgestellt. Die Autoren setzen den Ansatz nicht nur in der Lehre an der HAW-Hamburg sehr erfolgreich ein, sondern unterstützen damit auch kommerzielle Industrieprojekte.

### **1 Einleitung**

Ergänzend zu den in immer größerer Zahl angebotenen Prozessor IP-Cores für ASICs und FPGAs wie z.B. ARM [1, 2] und MicroBlaze [3] gibt es Anwendungsfelder, die von anwendungsspezifischen Prozessoren (ASPs) flexibler und kostengünstiger bedient werden können. Im Unterschied zu den Prozessor-IPs sind bei den ASPs die Instruktionssätze und daraus folgend die Device-Ressourcen nicht vorgegeben, sondern können im Hinblick auf die spezielle Anwendung selbst definiert werden.

Die Voraussetzung für den Entwurf von ASPs besteht in der exakten Analyse der Systemanforderungen (Hard- **und** Software) und den detaillierten Kenntnissen der inneren Strukturen von Architekturvarianten in Bezug auf Hardware-Ressourcenbedarf und Geschwindigkeit. Der sich daraus ergebende zusätzliche Entwicklungs- und Verifikationsaufwand ist aber nach unseren Erfahrungen durchaus vergleichbar zu dem Initialaufwand, der für die Integration eines neuen Prozessor IP-Cores zu leisten ist. Insbesondere für ASICs, die in großen Stückzahlen gefertigt werden, bieten ASPs jedoch Flächenvorteile, da nur die für den gewählten Instruktionssatz tatsächlich benötigten Prozessorkomponenten auf dem Silizium implementiert werden. Mit dem in diesem Bericht vorgestellten Framework-Ansatz besteht das Ziel, die Methoden des ASP-Entwurfs auf eine systematische Grundlage zu stellen. Dies betrifft insbesondere den Zusammenhang zwischen der Definition des Instruktionssatzes, der Realisierung der Daten- und Kontrollpfade und den sich daraus ergebenden Einschränkungen bei der Programmierung durch Kontroll- und Daten-Hazards.

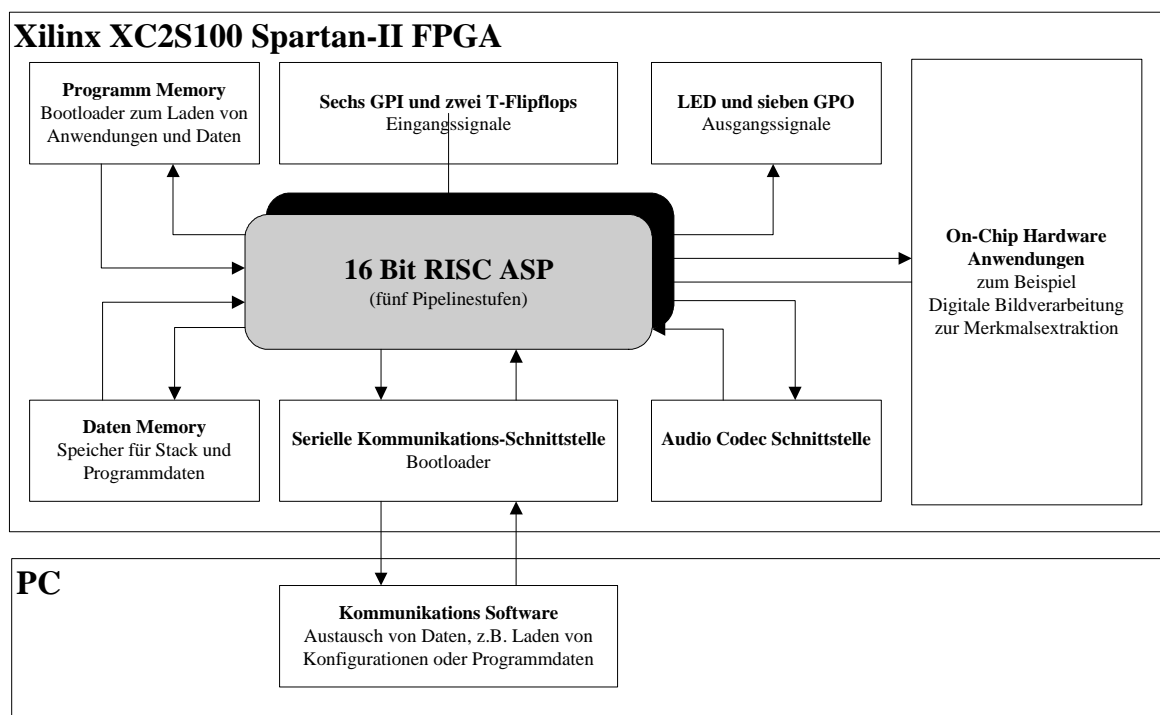
Im Rahmen zahlreicher interner Projekte sowie in einer Industrie-Kooperation hat sich gezeigt, dass es mit dem Framework-Ansatz in sehr kurzer Zeit möglich ist, die Anforderungen eines individuellen Instruktionssatzes durch FPGA- bzw. ASIC-Hardware abzubilden. Im Ergebnis konnte beim Redesign eines Monitor-ICs u.a. gezeigt werden, dass ein speziell für diesen Zweck entworfener RISC-Prozessor in diesem ASIC einen geringeren Flächenbedarf erfordert als das zuvor darin enthaltene digitale Rechenwerk [17].

Aus diesen Gründen ist der Entwurf und die Implementierung von ASPs von uns als Projektschwerpunkt für die Ausbildung im Fach Digitale Systeme gewählt worden, das als Pflichtveranstaltung bzw. Wahlpflichtfachveranstaltung für Studenten der Elektrotechnik im 6. und

7. Semester stattfindet. Neben dem vertieften Verständnis eines RISC-Prozessors wird so bei den Ingenieur-Studenten auch erreicht, dass sie die Methodik eines Systementwurfs vom Blockschaltbild über die Modellverifikation durch Timingsimulationen bis zum Nachweis der Hardware- und Softwarefunktionen an einer geschlossenen Aufgabenstellung erlernen. Seitens der Industrie wurde uns bestätigt, dass Absolventen, die auf diese Weise ausgebildet wurden, mit deutlich kürzerer Einarbeitungszeit in komplexe IC-Entwurfsprojekte integriert werden konnten.

Dieser Beitrag gibt eine Übersicht zu zwei unterschiedlich komplexen RISC basierten Prozessorarchitekturen [5-11] sowie zu integrierbaren Hardware- und Software-Modulen, die von uns und unseren Studenten konzipiert und entwickelt wurden [17, 20 – 26]. Mit dem Framework steht uns eine Entwicklungsplattform zur Verfügung, die wir in diverse System on Chip (SoC) F+E Projekte der Hochschule einbringen.

Der Aufsatz stellt im nachfolgenden Kapitel zunächst die verwendeten methodischen Ansätze vor. In Kapitel 3 wird ein Basis-Prozessor, sowie nachfolgend in Kapitel 4 ein Prozessor mit Interface zur C-Programmierung vorgestellt.



**Bild 1: Exemplarisches SoC Design mit ASP.**

## 2 Methodische Ansätze des skalierbaren Prozessorentwurfs

Der Hintergrund zur synthesesgerechten VHDL-Modellierung von CISC-Prozessoren wurde von den Autoren in ihrem Lehrbuch beschrieben [20]. Ergänzend dazu wurden verschiedenartige RISC-Konzepte in mehreren studentischen Projekten erarbeitet [21, 22, 23, 25]. Aus diesen Vorarbeiten entstand der Bedarf nach einer verbesserten Systematik für den gemeinsamen Entwurf des Instruktionssatzes und dem dazu gehörigen VHDL-Modell. Für die weiteren Prozessorentwicklungen ist folgender methodischer Ansatz zusammengestellt worden:

- Ausgangspunkt ist die bewährte RISC-Harvard-Architektur nach [5] u. [6].
- Partitionierung des Modells in Pipeline-Stufen als Komponenten mit synchronisierten Ein- und Ausgangssignalen.

- Abbildung der Komponenten:
  - Datenpfadanalyse und -realisierung auf RTL-Ebene mit VHDL-Designpatterns auf Basis eines Codingstyles gemäß IEEE 1076.6.
  - Zur Realisierung des Steuerpfades wurden die nachfolgenden Alternativen verglichen:
    - a) Opcode-Dekodierung parallel in jeder Pipelinestufe.
      - +: Übersichtlich planbar mit Multiplexertabellen.
      - : Es ergeben sich verlängerte Laufzeitpfade in der jeweiligen Stufe.
    - b) Opcode-Dekodierung in der jeweils vorausgehenden Pipelinestufe.
      - +: Keine verlängerten Laufzeiten in dem zu steuernden Datenpfad.
      - : Höherer HW-Aufwand für die Pipelineregister.
      - : Geringere Übersichtlichkeit der Strukturen.
    - c) Zentrale Opcode-Dekodierung für alle Datenpfade in der Decode-Pipelinestufe.
      - : Höherer HW-Aufwand für die Pipelineregister der Steuersignale.
      - : Keine Vorteile für die Übersichtlichkeit und die Datenpfad-Laufzeiten.

Die Umsetzung dieses Konzeptes für den skalierbaren Entwurf von RISC-Prozessoren mit den zugehörigen Hardware-Software-Schnittstellen wird nach folgenden Entwurfsrichtlinien durchgeführt:

- Ein zunächst einfacher Instruktionssatz wird in einer überschaubaren RISC-Architektur mit klar erkennbarem, getaktetem Phasenpipelining abgebildet. Das dazugehörige Top-Level VHDL Modell spiegelt die einzelnen Phasen in Form von Komponenten-Instanzen wider.
- Das Zusammenspiel zwischen den Instruktionen und Datenpfaden der Hardwarearchitektur muss klar erkennbar sein. Dies wird dadurch erleichtert, dass alle Prozessoraktionen innerhalb einer Pipelinestufe entweder kombinatorisch sind oder bei fallender Taktflanke erfolgen.
- Der Steuerpfad wird nach a) realisiert und ggf. zur Steigerung der Performance nach b) modifiziert.
- Um eine möglichst gute Abbildung des zu Grunde liegenden Blockdiagramms auf die VHDL- Modelle der Pipelinephasen zu erhalten, werden nur wenige Grundkomponenten der Digitaltechnik verwendet. Dazu gehören: ALU's, Multiplexer, Demultiplexer, Komparatoren, Register, getaktete Zähler und ggf. Speicher. Falls als Zielhardware FPGAs vorgesehen sind, lassen sich Multiplexer mit breiten Datenpfaden durch Tri-State-Treiber ersetzen.
- Für die in den einzelnen Pipelinestufen verwendeten (De-) Multiplexer wird tabellarisch erfasst, welche Instruktionen welche Datenpfade schalten.
- Als Unterstützung für die Softwareentwicklung muss für die verschiedenen Instruktionsgruppen eine Analyse potentieller Daten- und Kontrollfluss-Hazards [4] erfolgen.

Darauf aufbauend lassen sich nun die für die Anwendung spezifischen Instruktionen hinzufügen ( bzw. auch entfernen) wobei auf die folgenden Punkte geachtet werden muss:

- Hardware Ressourcen-Konflikte sind zu vermeiden. Ggf. sind zusätzliche Datenpfade hinzu zu fügen.
- Sofern das bisherige Pipelinekonzept beibehalten werden soll, erfolgt die Erweiterung der Hardware-Funktionalität durch Hinzufügen neuer VHDL-Prozesse und ggf. durch Modifikation der bereits vorhandenen Multiplexer-Prozesse.

- Zusätzliche I/O-Hardwarekomponenten können in den Adressraum eingefügt werden (Memory-Mapping) und erfordern einen zusätzlichen Adressdekoder-Prozess in der Speicherphase.

Ausgangspunkt für die bisher realisierten 16 Bit (Daten- und Adressbus) Hardwarearchitekturen ist eine modifizierte MIPS16-Architektur mit 5-stufiger Pipeline [5, 6]. Allerdings ist dieser Ansatz nicht bindend, vielmehr können problemlos weitere Pipelinestufen hinzugefügt werden, sollte es sich erweisen, dass die Ausführungsdauer einzelner Pipelinestufen nach dem Einfügen neuer Instruktionen zu lang geworden ist.

Nachfolgend sollen exemplarisch zunächst eine Basis-Implementierung und anschließend nachfolgend eine Implementierung mit C-Programmierschnittstelle vorgestellt werden.

### 3 Basis-Prozessor

Der Basis-Prozessor besitzt ein Register-File mit acht Registern (16 Bit), er bietet Steuerleitungen für externe Peripherie sowie Schnittstellen zur FPGA-internen Datenübergabe. Die Memory-Phase des Prozessors ist mit einem Datenbus und einem Adressdekoder so aufgebaut worden, dass parallele Hardware-Algorithmen als Module in die Memory-Map integriert werden können.

#### 3.1 Instruktionssatz der Basisarchitektur

Nachfolgend wird der Instruktionssatz mit 22 Instruktionen vorgestellt. Er besteht aus folgenden Instruktionsgruppen:

- Lade- und Speicheroperationen aus dem Datenspeicher.
- Arithmetisch-logische Operationen zwischen Registern bzw. Immediate-Konstanten.
- Sprungoperationen

Die Implementierung von Programmen erfolgt durch Assemblercode, der mit einem Tabellenassembler [12] in Maschinencode umgesetzt wird. Dieser HEX-File wird über die Initialisierung des On-Chip Xilinx-Block RAMs in den Synthese- und Implementierungsprozess des Prozessors eingebracht. Ausgangspunkt für die Erstellung des Instruktionssatzes der Basisarchitektur sind die folgenden Forderungen:

- Maximal 32 unterschiedliche Instruktionen (vgl. Tabelle 1)
- Jede Instruktion soll mit einem einzigen Buszyklus aus dem Programmspeicher gelesen werden können. Dies bedeutet, dass alle Instruktionen mit 16 Bit codiert werden müssen.
- Lade- und Speicheroperationen finden ausschließlich über Register statt (Load-Store Architektur).
- Existenz relativer Sprungoperationen (Branch), da die Sprungdistanz nur mit 11 Bit codiert werden kann, der Prozessor jedoch maximal mit 16 Bit adressieren soll. Bedingte Branch-Operationen werten den Inhalt von Flags aus.
- Arithmetisch, logische Operationen nur zwischen Registern (3-Register Operationen).
- Innerhalb der Instruktionsgruppen werden feste Bitfelder definiert, in denen z.B. Opcodes, die Quell- und Zielregister sowie die Konstanten codiert werden.

Die realisierten bedingten Sprungbefehle decken mit einer minimalen Anzahl den gesamten Integer-Zahlenbereich ab. Sie werten zwei Flags aus, die in der Execute-Phase allein aus dem Ergebnis des SUB-Befehls abgeleitet werden. Denn dekrementierende Schleifenzähler und Grenzwertabfragen geben auf Basis des SUB-Befehls ausreichend Code-Entwurfsspielraum.

Instruktion	OPCode	Beschreibung
NOP	0	No Operation.
ADD $R_{DEST}, R_{SRC1}, R_{SRC2}$	1	Addiere den Inhalt der beiden Quellregister.
SUB $R_{DEST}, R_{SRC1}, R_{SRC2}$	2	Subtrahiere den Inhalt der Quellregister $R_{SRC1} - R_{SRC2}$ .
CLR $R_{DEST}, R_{DEST}, R_{DEST}$	2	Löscht das Register $R_{DEST}$ mit dem SUB Befehl
AND $R_{DEST}, R_{SRC1}, R_{SRC2}$	3	Bitweise UND-Verknüpfung der Quellregisterinhalte.
OR $R_{DEST}, R_{SRC1}, R_{SRC2}$	4	Bitweise ODER-Verknüpfung der Quellregisterinhalte.
XOR $R_{DEST}, R_{SRC1}, R_{SRC2}$	5	Bitweise XOR-Verknüpfung der Quellregisterinhalte.
SLA $R_{DEST}, R_{SRC1}, count$	6	Arithmetisches Links-Schieben um $\langle count \rangle$ Bit-Positionen. Korrekte Vorzeichenerweiterung.
SRA $R_{DEST}, R_{SRC1}, count$	7	Arithmetisches Rechts-Schieben um $\langle count \rangle$ Bit-Positionen. Korrekte Vorzeichenwerweiterung.
MV $R_{DEST}, R_{SRC}$	8	Kopiere den Inhalt aus dem Quellregister $R_{SRC}$ in das Zielregister $R_{DEST}$
ADDIL $R_{DEST}, \langle constant \rangle$	9	Die Byte-Konstante wird mit dem Register $R_{DEST}$ addiert. Das MSByte bleibt unverändert.
ADDIH $R_{DEST}, \langle constant \rangle$	A	Die Byte-Konstante wird mit dem MSByte des Registers $R_{DEST}$ addiert, das LSByte bleibt unverändert.
LD $R_{DEST}, *R_{SRC}, \langle offset \rangle$	B	Adressiert den Datenspeicher mit dem Inhalt von $(R_{SRC} + \text{Address offset})$ und lädt das Datenwort an dieser Adresse in das Register $R_{DEST}$ .
ST $R_{SRC}, *R_{DEST}, \langle offset \rangle$	C	Adressiert den Datenspeicher mit dem Inhalt von $(R_{DEST} + \text{Address offset})$ und speichert den Inhalt von $R_{SRC}$ an dieser Adresse im Datenspeicher.
JMP $\langle address constant \rangle$	D	Springe zu der absoluten Adresse
BNE $\langle address offset \rangle$	E	Wenn Flag Zero $\neq 0$ , dann addiere den Zweierkomplement Offset zum PC.
BLT $\langle address offset \rangle$	F	Wenn FLAG Negative $\neq 1$ , dann addiere den Zweierkomplement Offset zum PC.
BGE $\langle address offset \rangle$	10	Wenn $(\text{FLAG Negative} \neq 0 \vee \text{Flag Zero} \neq 1)$ , dann addiere den Zweierkomplement Offset zum PC.
CALL $\langle address constant \rangle$	11	Funktionsaufruf; speichert den PC auf dem Stack und beschreibt den PC mit der absoluten Adresse.
RET	12	Rücksprung aus Funktion; lädt den PC mit Stack-Inhalt.
PUSH $R_{SRC}$	13	Adressiert den Datenspeicher mit dem Stack Pointer (post decrement) und schreibt den Inhalt des Quellregisters an diese Adresse.
POP $R_{DEST}$	14	Adressiert den Datenspeicher mit dem Stack Pointer (pre increment) und lädt das adressierte Datenwort ins Register $R_{DEST}$ .
MUL $R_{DEST}, R_{SRC1}, R_{SRC2}$	15	Multipliziert im Q-Format die oberen 12 Bit der Quellregister und speichert die oberen 16 Ergebnisbits inkl. 2 Vorzeichenbits ins Zielregister.

**Tabelle 1: Instruktionssatz der Basisarchitektur**

### 3.2 Architektur

Bild 2 zeigt die fünfstufige RISC-Pipeline, die stark an die MIPS16-Architektur [5, 6] angelehnt ist. Zu erkennen sind die Pipelinestufen:

- Instruction Fetch → Holen der Instruktion aus dem Instruktionsspeicher
- Instruction Decode → Dekodieren der Instruktion, Holen der Registeroperanden und Adressberechnungen (ALU 1)
- Execute → Ausführen der arithmetisch-logischen Instruktion und Stack-Pointeraktualisierung
- Memory Access → Schreib- / Lesezugriff auf den Datenspeicher, Laden des Programmspeichers, Datentransfer zur Codec-Schnittstelle, zu GPI/Os und zur seriellen PC-Schnittstelle
- Write Back → Zurückschreiben der Ergebnisse aus der Execute-Phase ins Register-File

Die Phasenregister IF/ID, ID/EX, EX/MEM, und MEM/WB bilden jeweils den Ausgang der Pipelinestufen. Dieser Ansatz hat den Vorteil, dass die einzelnen Stufen während der Entwicklung sehr einfach testbar sind, denn beim Entwurf der jeweiligen Testumgebung der jeweils nachfolgenden Stufe kann davon ausgegangen werden, dass alle Eingangssignale synchron sind.

In dieser Architektur werden der Programmzähler, der Datenspeicher, alle externen Schnittstellen und ebenso das Registerfile synchron bei fallender Taktflanke getriggert, um die jeweiligen Daten innerhalb eines Taktzyklus' bereitstellen zu können. Die Leseseite des Programmspeichers kann als in das Pipelineregister integriert angesehen werden, sodass das Pipelineprinzip nicht verletzt wird. Die Befehlsdekorierer der einzelnen Pipelinestufen sind der Übersichtlichkeit halber nicht dargestellt. Innerhalb der einzelnen Stufen werden die folgenden Aktionen ausgeführt:

- **IF:** Inkrementierung des Programmzählerregisters bei fallender Taktflanke. Für auszuführende Sprünge synchrones Laden der Zieladresse. Der Programmspeicher ist als Dual-Port On-Chip RAM implementiert, sodass ein Bootloader-Programm die Anwender-Instruktionen über Store-Befehle in den Programmspeicher laden kann. Da die Schreibseite des Programmspeichers logisch zur MEM-Phase gehört, wird sie mit der negativen Taktflanke gesteuert
- **ID:** Dekodierung der Instruktion und Selektion der Operanden-Quellregister. Die Verzweigung bei Sprüngen wird entschieden und die Zieladresse wird berechnet. Eine vorzeichenrichtige Erweiterung von Konstanten findet hier statt, um die Laufzeitpfade der Execute-Phase zu verkürzen.  
In dem ID-Funktionsblock erfolgt auch das Abspeichern eines aus der WB-Phase kommenden Datenworts im Zielregister.
- **EX:** Die ALU besteht aus parallelen Funktionblöcken, deren Operanden und Ergebnisse über Busse bereitgestellt bzw. gesammelt werden. Durch den Einsatz von Tri-State Treibern anstelle von breiten Multiplexern ergeben sich deutliche Logikeinsparungen. Der Stack-Pointer liegt in der EX-Phase, da er so die Addierer-Hardware der ALU nutzen kann. Die Reduzierung des Multipliziererergebnisses von 24 auf 16 Bit erfolgt mit Betragsschneiden, wobei negative Ergebnisse zum betragsmäßig kleineren Wert gerundet werden.

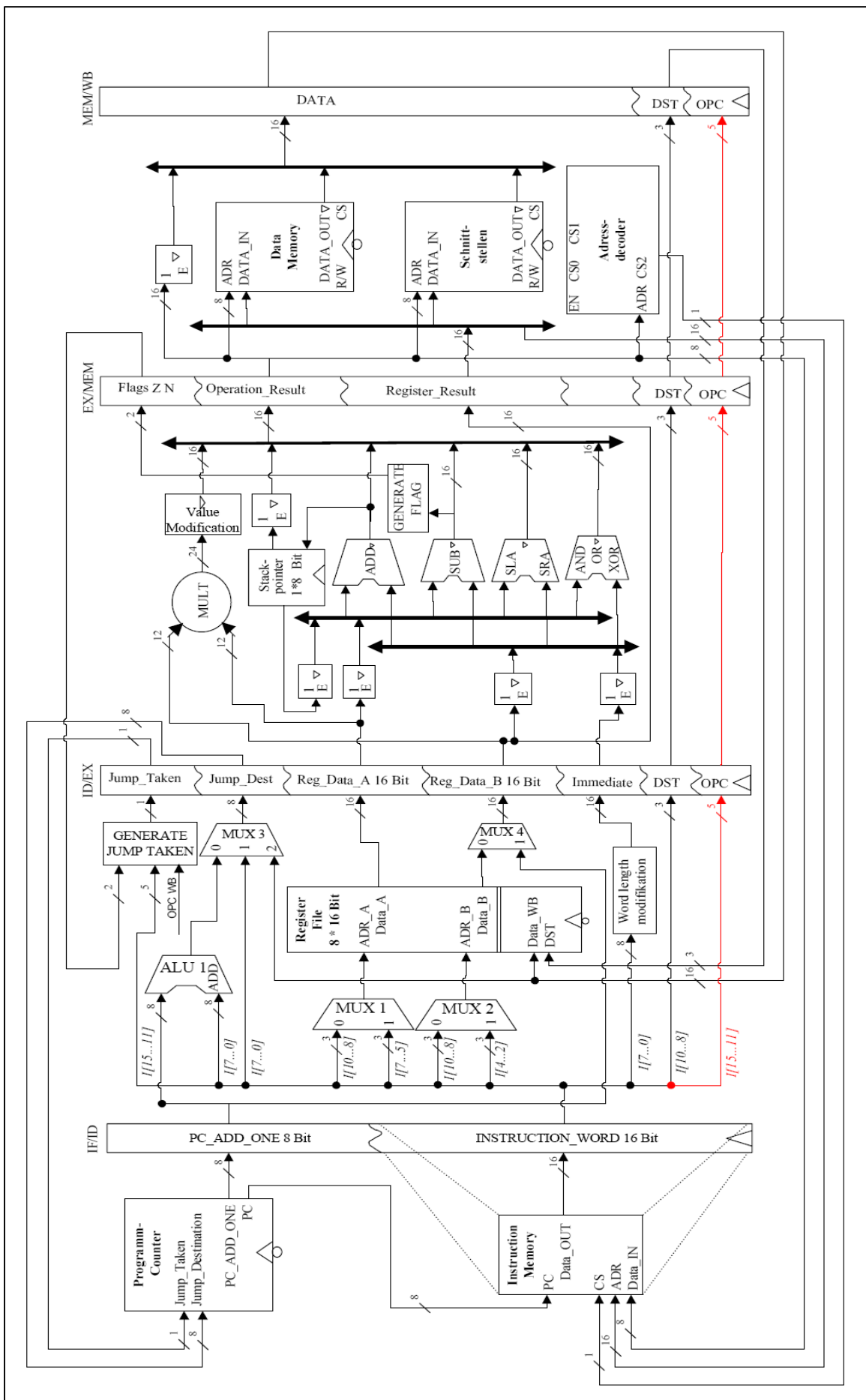


Bild 2: Aufbau des Basis-Prozessors mit Harvard Architektur und 1kB Speicher [26].

- **MEM:** Auch in dieser Phase realisieren Tri-State Treiber die Multiplexeraufgaben für die Datenpfade zum Abspeichern bzw. Laden von Registerinhalten aus dem Datenspeicher. Hierfür wird das Ergebnis der ALU-Operation als Speicheradresse verwendet und der Registerinhalt wird bei Store-Befehlen auf den Datenbus (DATA\_IN) gelegt. Bei Ladebefehlen wird der Inhalt der adressierten Speicherzelle über den Bus DATA\_OUT in das MEM/WB Pipelineregister geschrieben. Der in Bild 2 dargestellte Block „Schnittstellen“ fasst alle Anwenderschnittstellen (Codec-, serielles PC-Interface und GPI/O) zusammen, die mit dem Adressdekoder selektiert werden (vgl. Kap. 3.3). Die MEM-Phase ist direkt mit dem Programmspeicher gekoppelt, sodass ein Bootloader-Modul die Anwender-Software über die serielle PC-Schnittstelle mittels Load- und Store-Befehlen laden kann. Alle anderen Ergebnisse aus den arithmetisch/logischen Operationen gehen parallel zu den Interfaceblöcken direkt in das MEM/WB Pipelineregister.
- **WB:** Diese Phase enthält keine dedizierte Hardware, da die entsprechende Selektionslogik bereits in der ID-Phase enthalten ist. Zu ladende Daten oder weitergeleitete ALU-Ergebnisse werden mit der negativen Taktflanke durch diese Hardware der ID-Phase im Register-File gespeichert. Bei der folgenden steigenden Flanke stehen aktualisierte Registerinhalte zur sofortigen Bearbeitung durch die EX-Phase im ID/EX Pipelineregister zur Verfügung.

### 3.3 Hardware- und Software-Anwenderschnittstellen

Nachfolgend werden die Schnittstellen des beschriebenen RISC-Prozessors vorgestellt, über die der Anwender mit dem Prozessor kommuniziert.

#### 3.3.1 Serielles PC-Interface

Für den bidirektionalen Datentransfer zwischen einem PC und dem Basis-Prozessor ist ein serielles Kommunikationsmodul geschaffen worden, das auf der einen Seite aus einem VHDL-Modul und auf der anderen Seite aus einer COM-Port PC-Software besteht. Das VHDL-Modul ist über den Adressdekoder der MEM-Phase in den Adressraum des Prozessors eingebunden und kann somit über die Load- und Store-Befehle angesprochen werden. Der serielle Empfänger enthält eine Paritätsprüfung des Bitstrings und eine Majoritätsbewertung der dreifach abgetasteten Bits zur Störunterdrückung. Übertragungsraten stehen bis 96 kBaud zur Verfügung.

#### 3.3.2 Bootloader-Modul

Ein Bootloader-Softwaremodul sorgt dafür, dass dem Basis-Prozessor nach einem Hardware-Reset über die serielle Schnittstelle ein neues Programm eingespielt wird. Im Pollingbetrieb kommuniziert der Bootloader mit dem seriellen Interface, das mit zwei Byte-Transfers eine Instruktion liefert. Empfangene Instruktionen werden zuerst in das Register-File geladen und anschließend über den Schreibeingang im Dual-Port-Programmspeicher abgelegt. Programmänderungen lassen sich so schnell durchführen, ohne dass eine komplette Synthese und Implementierung des gesamten Prozessors erforderlich wird.

#### 3.3.3 Audio-Codec Interface

Zur Demonstration der Performance des Basis-Prozessors sind FIR-Filter für Audioanwendungen in Assembler codiert worden. Der Zugriff auf die ADU/DAU-Kanäle des auf dem FPGA-Board verfügbaren Codec's erfolgt dabei über ein Interface, das die synchronen, seriellen Codec-Telegramme im Pollingbetrieb steuert. Die 48 kHz Abtastrate des Codec bestimmt das der Software zur Verfügung stehende Zeitintervall zur Aktualisierung eines Ausgangswertes.

### 3.4 Implementierungsergebnisse

Der Basis-Prozessor nach Bild 2 wurde mit einem Xilinx Spartan-II FPGA XC2S1005TQ144 implementiert und mit 40 MHz betrieben. Als Hardware-Plattform diente eine Kombination der Boards XSA100 und XSTend II der Firma XESS. Die benötigten FPGA Hardware-Ressourcen teilen sich folgendermaßen auf: 656 Slices (54%), D-FFs 488 (20%), Look up Tabellen 944 (39%), Tri-State Treiber 240 (18%). Der Block "Schnittstellen" in der MEM-Phase trägt dazu nur etwa 14% bei. Die maximal zulässige Taktfrequenz von 41,6 MHz wird durch den Laufzeitpfad zwischen dem Datenspeicher und dem MEM/WB-Register in der MEM-Phase festgelegt. Dort wird von der negativen zur positiven Taktflanke eine minimale Laufzeit von 12.008 ns benötigt. Den längsten Laufzeitpfad zwischen zwei positiven Taktflanken bildet der Multiplizierer in der EX-Phase mit 23.989 ns. Für die Erprobung von FIR-Filtern sind unter diesen Timingbedingungen und mit optimiertem Assemblercode maximal realisierbare Filterordnungen berechnet worden: Ein Einzelfilter lässt sich bis zur Ordnung von 56 realisieren. Eine steuerbare Kombination von Tief-, Hoch- und Bandpassfiltern kann bei Berücksichtigung der Filtersymmetrie bis zur Ordnung von je 42 realisiert werden.

## 4 Prozessor mit C-Programmierschnittstelle (HAWRISC1)

Obwohl die Hardwarearchitektur von ASPs auf spezielle Aufgaben zugeschnitten ist, und sich die dazu gehörige Software häufig sehr effizient in Assemblercode entwickeln lässt, besteht seitens vieler Softwareentwickler dennoch der Wunsch, die Programmierung in einer Hochsprache, z.B. C vornehmen zu können. In einer Anforderungsanalyse wurden die in [14] vorgestellten Cross-Compiler mit frei programmierbarer Assembler-Schnittstelle (Backend) in Bezug auf ihre Verwendbarkeit verglichen. Im Ergebnis entschieden wir uns für den lcc-Compiler [13], der eine einfachere zu programmierende Assembler-Schnittstelle bietet. Die für diesen Compiler erforderlichen Instruktionen führten zu einer grundsätzlichen Neukonzeption des Instruktionssatzes.

### 4.1 Instruktionssatz des HAWRISC1

Die folgenden Gesichtspunkte wurden bei der Definition des HAWRISC1-Instruktionssatzes berücksichtigt:

- Eine mit einem Registersatz von nur 8 Registern ausgestattete Hardware ist für eine effektive C-Code Compilierung ungeeignet. Daher wird das Registerfile auf 16 Register erweitert. Davon wird in Übereinstimmung mit [16] r0 als Null-Register (Konstante 0) und r15 als Stack-Pointer (sp) verwendet.
- Da in diesem Konzept für 3-Register Operationen bereits 12 Bit erforderlich sind, ein Umfang von 16 Instruktionen aber nicht ausreicht, muss das in Tabelle 1 vorgestellte Konzept mit fest definierten Bitfeldern für die 16-Bit breiten Instruktionen verlassen werden.
- Das Compiler-Backend erfordert diverse Operationen mit 16-Bit Immediate Konstanten, die sich nicht in einem Instruktionssatz codieren lassen.
- Zur einfachen Realisierung der im C-Code verwendeten Bedingungen, z.B. für den Schleifenabbruch ist es sinnvoll, flexiblere bedingte Branch-Operationen einzuführen.

Als Konsequenz wurde der in Tabelle 2 aufgelistete Instruktionssatz definiert, der sich im Vergleich zu dem in Tabelle 1 angegebenen wie folgt unterscheidet:

- Es existieren Instruktionen mit *einem* Adresszugriff auf den Programmspeicher, sowie solche, die *zwei* Buszyklen erfordern.
- Die Codierung des Instruktionstyps erfolgt nicht mehr allein in den oberen Bitfeldern 15..12, sondern es wird - je nach Instruktionsgruppe - eine Subcodierung in den Bitfeldern 11..8 bzw. 3..0 vorgenommen.

Daraus entsteht ein im Vergleich zur Basisarchitektur deutlich größerer Hardwareaufwand zur Befehlsdekodierung und ein entsprechend längerer Signallaufzeitpfad dieser Phase, der bei der Implementierung dieser Architektur zu analysieren sein wird.

Bit Fields				Instruction	Effect
15..12	11..8	7..4	3..0		
0	0	0	0	<b>nop</b>	no operation
1	rd	ra	rb	<b>add</b> rd, ra, rb	$r[rd] ? r[ra] + r[rb]$
2	rd	ra	rb	<b>sub</b> rd, ra, rb	$r[rd] ? r[ra] - r[rb]$
3	rd	ra	rb	<b>and</b> rd, ra, rb	$r[rd] ? r[ra] \wedge r[rb]$
4	rd	ra	rb	<b>or</b> rd, ra, rb	$r[rd] ? r[ra] \vee r[rb]$
5	rd	ra	rb	<b>xor</b> rd, ra, rb	$r[rd] ? r[ra] \oplus r[rb]$
6	rd	ra	rb	<b>mul</b> rd, ra, rb	$r[rd] ? (r[ra] * r[rb]) \gg 15$
7	rd	ra	0	<b>addi</b> rd, ra, imm16	$r[rd] ? r[ra] + \text{imm16}$ ; see note 1)
7	rd	ra	1	<b>subi</b> rd, ra, imm16	$r[rd] ? r[ra] - \text{imm16}$ ; see note 1)
7	rd	ra	2	<b>andi</b> rd, ra, imm16	$r[rd] ? r[ra] \wedge \text{imm16}$ ; see note 1)
7	rd	ra	3	<b>ori</b> rd, ra, imm16	$r[rd] ? r[ra] \vee \text{imm16}$ ; see note 1)
7	rd	ra	4	<b>xori</b> rd, ra, imm16	$r[rd] ? r[ra] \oplus \text{imm16}$ ; see note 1)
7	rd	0	5	<b>law</b> rd, ra, imm16	$r[rd] ? *(r[ra] + \text{imm16})$ see note 2)
7	rb	0	6	<b>saw</b> rb, ra, imm16	$*(r[ra] + \text{imm16}) ? r[rb]$ see note 2)
7	rd	imm4	7	<b>srli</b> rd, rd, imm4	$r[rd] ? r[rd] \gg \text{imm4}$ (leading zeros)
7	rd	imm4	8	<b>srai</b> rd, rd, imm4	$r[rd] ? r[rd] \gg \text{imm4}$ (sign extension)
7	rd	imm4	9	<b>slli</b> rd, rd, imm4	$r[rd] ? r[rd] \ll \text{imm4}$ (zero padding)
7	rd	0	A	<b>mvk</b> rd, #imm16	$rd = \text{imm16}$ ; see note 1)
9	rd	imm4	ra	<b>lw</b> rd, imm4, (ra)	$r[rd] ? *(r[ra] + \text{imm4})$ see note 3)
A	rd	imm4	ra	<b>sw</b> rd, imm4, (ra)	$*(r[ra] + \text{imm4}) ? r[rd]$ see note 3)
B	0	disp8		<b>br</b> label	$pc ? pc + \text{sign\_ext}(\text{disp})$
B	1	disp8		<b>beq</b> label	if ( $=$ ) $pc ? pc + \text{sign\_ext}(\text{disp8})$
B	2	disp8		<b>bne</b> label	if ( $\neq$ ) $pc ? pc + \text{sign\_ext}(\text{disp8})$
B	3	disp8		<b>bc</b> label	if (carry) $pc ? pc + \text{sign\_ext}(\text{disp8})$
B	4	disp8		<b>bnc</b> label	if (!carry) $pc ? pc + \text{sign\_ext}(\text{disp8})$
B	5	disp8		<b>bv</b> label	if (overflow) $pc ? pc + \text{sign\_ext}(\text{disp8})$
B	6	disp8		<b>bnv</b> label	if (!overflow) $pc ? pc + \text{sign\_ext}(\text{disp8})$
B	7	disp8		<b>blt</b> label	if ((signed) $<$ ) $pc ? pc + \text{sign\_ext}(\text{disp8})$
B	8	disp8		<b>bgt</b> label	if ((signed) $>$ ) $pc ? pc + \text{sign\_ext}(\text{disp8})$
B	9	disp8		<b>ble</b> label	if ((signed) $\leq$ ) $pc ? pc + \text{sign\_ext}(\text{disp8})$
B	A	disp8		<b>bge</b> label	if ((signed) $\geq$ ) $pc ? pc + \text{sign\_ext}(\text{disp8})$
C	0	0	0	<b>call</b> function	$r[sp] ? pc; sp?sp-1; pc ? \text{imm16}$ ; see note 4)
C	0	0	1	<b>ret</b>	$sp ? sp+1; pc ? r[sp];$

#### Notes:

- 1) The 16 bit immediate operand must be coded directly behind the instruction. law- and saw-operations allow direct loading / storing of data from / to data-memory addresses.
- 2) law (load address word) loads the content of a memory address which is given by the sum of the value in register ra and the 16-bit immediate operand to register rd. Correspondingly saw (store address word) stores the content of register rb to a memory address which is given by the content of ra and imm16. Both instructions require two words.
- 3) lw (load word) loads the content of a memory address which is given by the sum of the value in register ra and the 4-bit immediate operand to register rd. Correspondingly sw (store word) stores the content of register rd to a memory address which is given by the content of ra and imm4. Both instructions require a single word.
- 4) The 16 bit target address must be coded directly behind the call-instruction. It allows function addresses within a 16-bit address range.

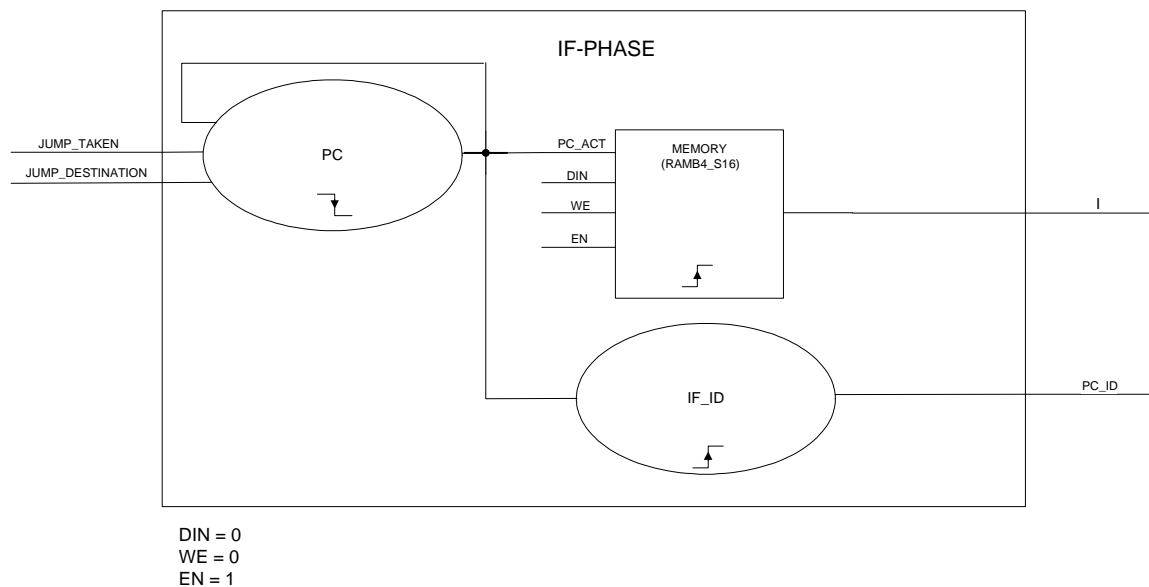
### Tabelle 2: Instruktionssatz des HAWRISC1.

#### 4.2 Architekturmodifikationen

Im Gegensatz zu der Funktionsblockbeschreibung in Kap. 3.2 soll nachfolgend die VHDL-Struktur der Hardwarearchitektur des HAWRISC1 durch Komponenten (Rechtecke) sowie Prozesse bzw. nebenläufige Anweisungen (Ovale) beschrieben werden. Bei allen Prozessen handelt es sich um VHDL-Designpatterns auf RTL-Ebene. Auf der obersten Entwurfsebene werden die einzelnen Pipelinestufen als VHDL-Komponenten wie folgt modelliert:

➤ **IF** (vgl. Bild 3):

Die IF-Phase besteht aus einem Programmzähler (PC), der als ladbarer Zähler mit fallender Flanke modelliert ist, dem Programmspeicher (MEMORY), der hier als On-Chip Xilinx-Block RAM Komponente (RAMB4\_S16) instanziiert wurde, sowie dem Pipeline-Register (IF\_ID).

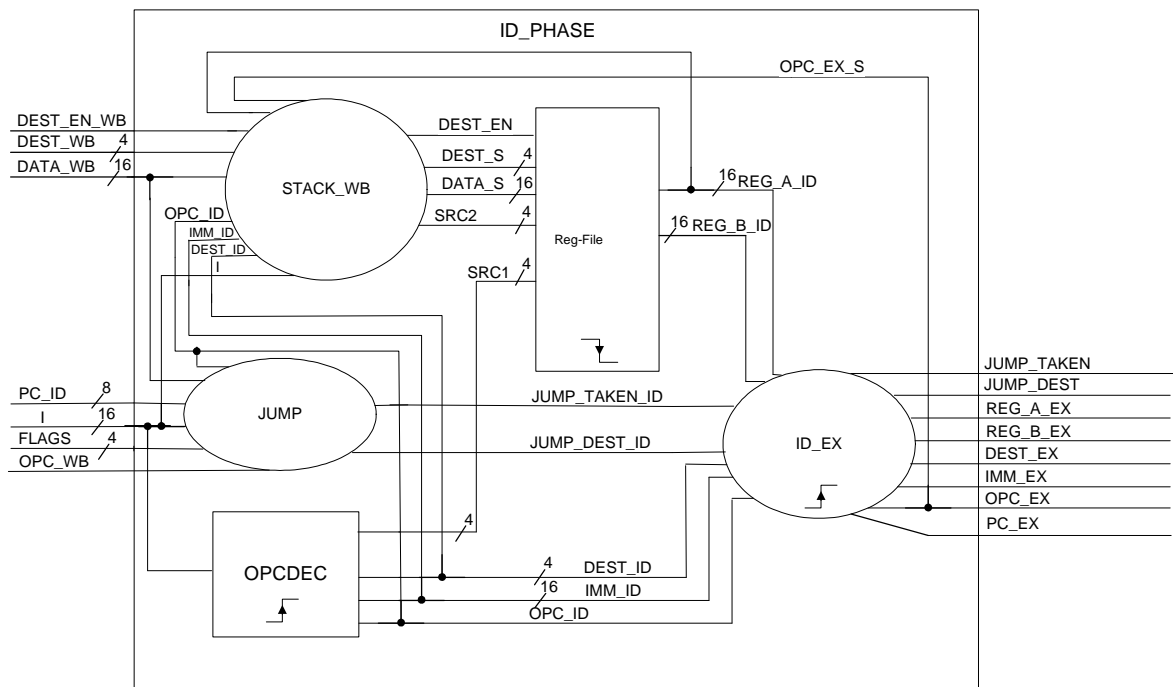


**Bild 3: VHDL-Struktur der IF-Phase.**

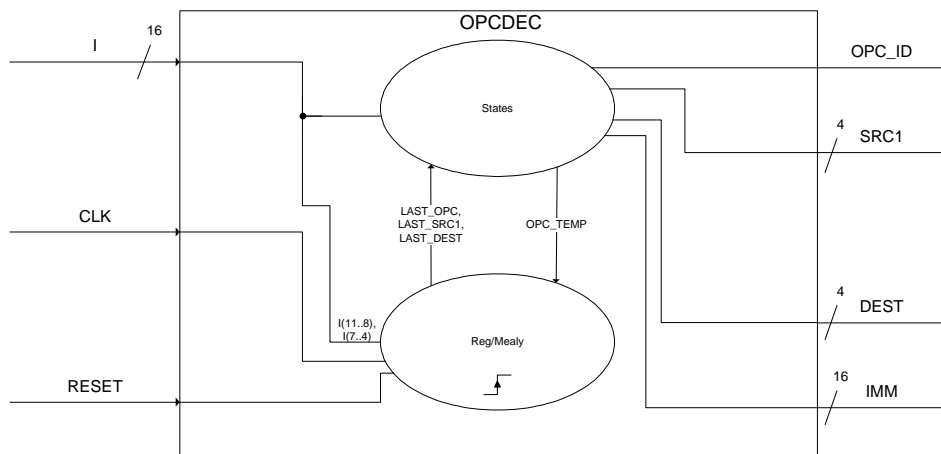
➤ **ID** (vgl. Bild 4):

Die ID-Phase ist im Vergleich zu dem in Kap. 3 vorgestellten Prozessor deutlich aufwändiger:

- Das Register-File (Reg-File) wurde als parametrisierbare Komponente aus früheren Projekten übernommen.
- Der Prozess STACK\_WB regelt die Lese- und Schreib-Zugriffssteuerung auf das Register-File: Je nach Instruktion werden die Datenpfade so geschaltet, dass Registerinhalte abgerufen bzw. Ergebnisse aus der MEM-Phase (s.u.) im Zielregister abgelegt werden können.
- Der Prozess JUMP wertet die von der letzten Instruktion generierten, sich in der EX-Phase (s.u.) befindlichen Flaggen aus und generiert das JUMP\_TAKEN\_ID-Signal, wenn sich eine Sprunganweisung in der ID-Phase befindet. Außerdem enthält der Prozess eine Adress-ALU zur Sprungzielberechnung (JUMP\_DEST\_ID).
- Der Prozess ID\_EX speichert die Ausgangssignale der ID-Phase taktsynchron im Pipeline-Register.
- OPCDEC (vgl. Bild 5) ist ein als Komponente modellierter 2-Prozess Zustandsautomat mit Mealy-Verhalten, der die Datenpfade für Ein- und Zwei-Wort Instruktionen unterschiedlich schaltet: Bei Ein-Wort Instruktionen werden die Operandenfelder der Instruktion sofort an den Ausgang weiter geleitet, für Zwei-Wort Instruktionen werden die Operandenfelder zunächst für einen Takt zwischen gespeichert und von dem Automaten ein NOP in die Pipeline injiziert. Beim zweiten Takt wird der während des ersten Takts dekodierte Opcode zusammen mit allen Operanden weiter geleitet.



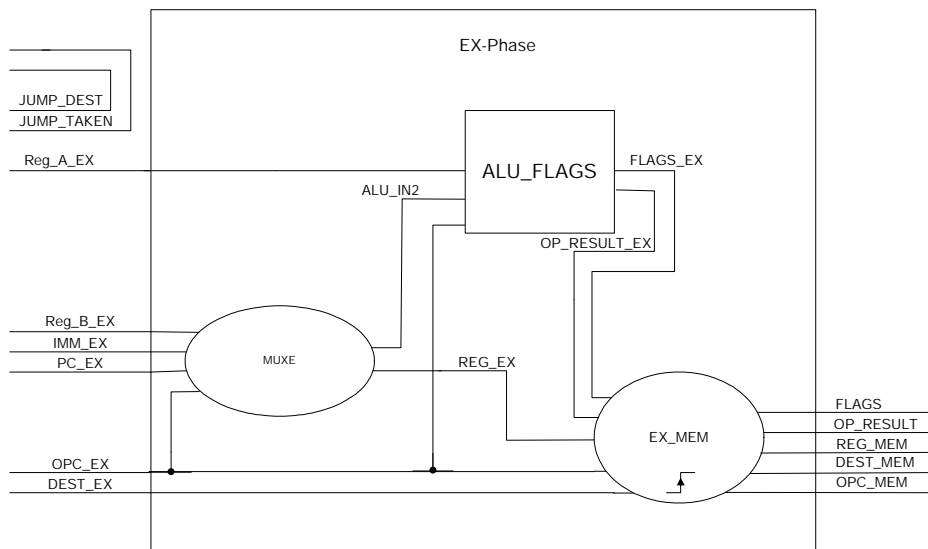
**Bild 4: VHDL-Struktur der ID-Phase.**



**Bild 5: VHDL-Struktur des OPCODE Zustandsautomaten.**

➤ **EX** (vgl. Bild 6):

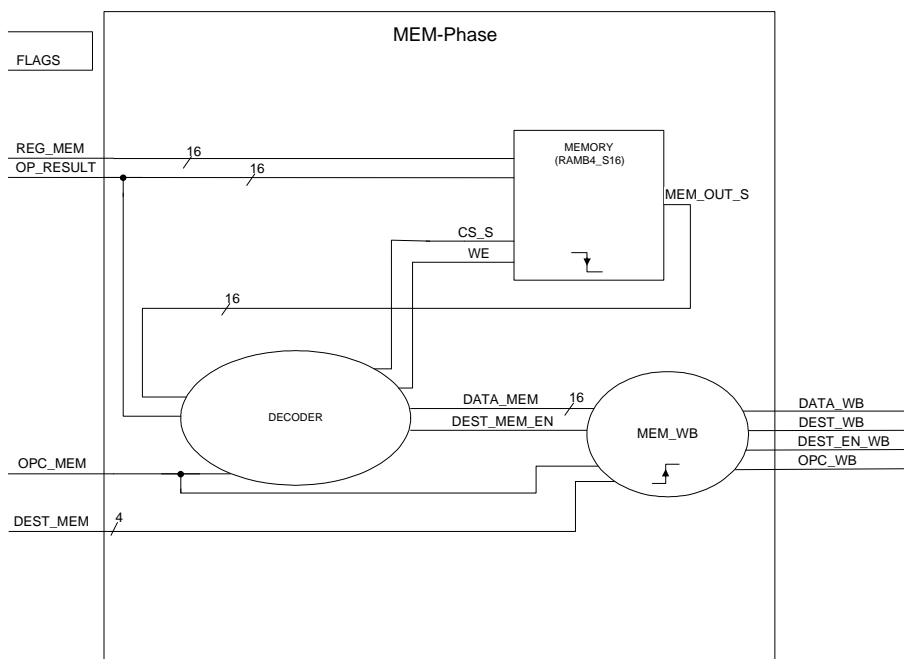
Die EX-Phase enthält als Kernelement eine 16-Bit-Alu mit integrierter Flaggen-Auswertung (ALU\_FLAGS). In dem MUXE-Prozess werden die Datenpfade so geschaltet, dass in der ALU entsprechend dem OPCODE entweder Register-Register Operationen berechnet werden oder solche, die einen Immediate-Operanden verwenden. Der EX\_MEM-Prozess enthält das Pipeline-Register dieser Stufe.



**Bild 6: VHDL-Struktur der EX-Phase.**

➤ **MEM** (vgl. Bild 7):

Während der MEM-Phase werden die Daten aus dem Datenspeicher gelesen bzw. die zu schreibenden Daten dort abgelegt. Die Komponente MEMORY stellt ein Xilinx-Block RAM (RAMB4\_S16) dar, welches durch OP\_RESULT adressiert wird. Die zu schreibenden Daten (Store-Befehle) liegen auf dem REG\_MEM-Bus und die zu lesenden Daten werden über MEM\_OUT\_S an einen Multiplexer Prozess weiter geleitet. Dieser Prozess (DECODER) dekodiert aus der aktuellen Instruktion in dieser Phase auch die Steuersignale für den Datenspeicher. Der Prozess MEM\_WB modelliert das Pipeline-Register der MEM-Phase.



**Bild 7: VHDL-Struktur der MEM-Phase.**

### 4.3 Hardware-Aufwand und Performance

Der vollständige HAWRISC1 lässt sich in einem Xilinx Spartan-2 Baustein mit 50k Gatter-äquivalenten (XC2S50) bei 90% Hardwareauslastung implementieren. Sofern auf die Schiebepfeile mit variabler Schiebeweite (Barrel-Shifter) verzichtet wird, reduziert sich der Ressourcenverbrauch in diesem FPGA auf ca. 75%. Ohne spezielle Timing-Constraints anzulegen, lässt sich der Prozessor in diesem Baustein mit 20 MHz betreiben. Der kritische Pfad liegt in der ID-Phase: Dort steht für die Freigabe zum Schreiben des Stack-Pointers (R15 im Registerfile) nur die halbe Taktperiode zur Verfügung. Als einfacher Lösungsansatz kann das Registerfile bei steigender Taktflanke getaktet werden. Dies muss allerdings durch das Einfügen eines zusätzlichen NOPs erkauft werden, wenn ein schreibender und lesender Registerzugriff nacheinander folgen sollen. Durch diese Maßnahme lässt sich die maximale Taktfrequenz jedoch von 20MHz auf 33MHz erhöhen.

## 5 Ausblick

Die in diesem Bericht beschriebene Entwurfssystematik hat sich als sehr erfolgreich beim zügigen Entwurf von ASP's heraus gestellt. Nach den Erfahrungen der Autoren kann eine typische Design-Zykluszeit, angefangen bei der Definition des Instruktionssatzes bis zum FPGA-Prototypen, mit ca. 2 Mann-Monaten abgeschätzt werden. Aus diesem Grunde wird die Methode auch seit einigen Jahren in Semesterprojekten der HAW-Hamburg mit Erfolg eingesetzt.

In einer Weiterentwicklung des Projektansatzes ist geplant, die Methodik auf eine höhere Abstraktionsebene zu portieren: Der Entwicklungsprozess soll noch weiter beschleunigt werden, indem VHDL gegen die Hardwarebeschreibungssprache Handel-C [15] ausgetauscht wird. Als konkrete Anwendung der ASPs ist die Realisierung von intelligenten Framegrabbern mit einer Bibliothek von Bildvorverarbeitungsalgorithmen geplant.

## 6 Literatur

- [1] St. Furber: ARM system-on-chip architecture. Addison Wesley, 2<sup>nd</sup> ed. 2000
- [2] ARM: The Information Quarterly. [www.convergencepromotions.com](http://www.convergencepromotions.com)
- [3] Xilinx: MicroBlaze Architecture. [www.xilinx.com/ipcenter/processor\\_central/microblaze](http://www.xilinx.com/ipcenter/processor_central/microblaze)
- [4] C. Siemers, Prozessorbau, Carl Hanser Verlag, 1999
- [5] J.L. Hennessy, D.A. Patterson, Computer Architecture, A Quantitative Approach, Morgan Kaufmann Publishers, 3<sup>rd</sup> ed. 2003
- [6] D. A. Patterson; J. L. Hennessy: Computer organisation & Design. The Hardware/Software Interface. Morgan Kaufmann Publishers, 2<sup>nd</sup> ed. 1998
- [7] W. Stallings: Computer Organization & Architecture. Designing for Performance. Prentice Hall, 6<sup>th</sup> ed. 2003
- [8] S. Lee: Design of Computers and other Complex Digital Designs. Prentice Hall 2000.
- [9] Ch. Märtin: Rechner Architekturen. CPUs, Systeme, Software-Schnittstellen. Carl Hanser Verlag 2001.
- [10] A. S. Tanenbaum; J. Goodman: Computerarchitektur. Strukturen, Konzepte, Grundlagen. Pearson Studium 2001.
- [11] W. Wolf: Computer as Components. Principles of Embedded Computing System Design. Morgan Kaufmann Publishers 2001.
- [12] H.P. Hohe: HASM, Tabellenkonfigurierbarer Makro-Assembler mit integriertem Linker, [hoh@iis.fhg.de](mailto:hoh@iis.fhg.de)
- [13] C. Fraser, S. Hanson: A Retargetable C-Compiler: Design and Implementation, Addison-Wesley, 1995.
- [14] R. Leupers, C. Marwedel: Retargetable Compiler Technology for Embedded Systems Tools and Applications, Kluwer Academic Publishers 2001
- [15] Handel-C is a C-based hardware description language, see [www-celoxica.com](http://www-celoxica.com)
- [16] Gray Research LLC, Porting lcc, [www.fpgacpu.org](http://www.fpgacpu.org), 2003
- [17] E. Unger: Flächenoptimierung eines digitalen Rechenwerks für einen Deflection Controller durch Implementierung einer RISC-Architektur, FB Elektrotechnik und Informatik der HAW-Hamburg, März 2003
- [18] J. Reichardt, B. Schwarz: A Framework Design of RISC Processors for FPGA-Based Rapid Prototyping, submitted to FPL 2003, Lisbon
- [19] J. Reichardt, B. Schwarz: Ein Framework zum Entwurf Applikationsspezifischer RISC Prozessoren (ASPs), eingereicht zur "Embedded Systems Conference", München 2004

- [20] J. Reichardt, B. Schwarz: VHDL-Synthese, Entwurf digitaler Schaltungen und Systeme, 3. Auflage, Oldenbourg 2003
- [21] O. Schwerin, H. Raap, M. Hehn, P. Ewald: Entwurf eines digitalen Signalprozessors auf FPGA-Basis, FB Elektrotechnik und Informatik der HAW-Hamburg, Projektdokumentation im WS 2002/2003.
- [22] L. Beseke, N. Breuhahn, A. Kühn, S. Peemöller: Entwurf eines digitalen Signalprozessors auf FPGA-Basis, FB Elektrotechnik und Informatik der HAW-Hamburg, Projektdokumentation im WS 2002/2003.
- [23] B. Huebener, S. Johannes, O. Tetzlaff: Entwicklung eines einfachen RISC-Prozessors auf FPGA-Basis, FB Elektrotechnik und Informatik der HAW-Hamburg, Projektdokumentation im SS 2003.
- [24] J. Graband: Implementation of a FPGA based DCT / IDCT demonstrator for the evaluation of Handel-C, Diplomarbeit im FB Elektrotechnik und Informatik der HAW-Hamburg, November 2003.
- [25] D. Dajka, O. Höltnke, H. Wulff: Entwicklung einer einfachen RISC-CPU auf FPGA-Basis, FB Elektrotechnik und Informatik der HAW-Hamburg, 2003.
- [26] B. Ajvazi: Entwicklung eines anwendungsspezifischen RISC-Prozessors für eine FPGA-Codex Plattform. Diplomarbeit im FB Elektrotechnik und Informatik der HAW Hamburg, Januar 2004.