ssfp16

Small Simple FPGA 16-bit Microprocessor

Rechtliches

Sämtliche Quellcodes dieses Projekts unterliegen der GNU General Public License (inoffizielle deutsche Übersetzung). Dies trifft auch für die VHDL-Beschreibung des Prozessors zu, welche ebenfalls zu Software im Sinne der GNU GPL gerechnet wird. Ich behalte mir vor, die von mir erstellten Teile zu einem späteren Zeitpunkt auch unter andere Lizenzen zu stellen. Dies gilt selbstverständlich nicht für ggf. zukünftig von Dritten gemäß GNU GPL entwickelte Module oder Verbesserungen.

Dieses Projekt ist in erster Linie zu Demonstrations- und Lehrzwecken gedacht. Eine patentrechtliche Prüfung konnte daher nicht vorgenommen werden. Da ich jedoch keinerlei Vorlagen für dieses Projekt nutzte und trotz meiner nur unwesentlich über Studieninhalte hinausgehenden Kenntnisse bezüglich Mikroprozessor-Architektur an keinem Punkt mehr als wenige Stunden nachdenken musste, gehe ich davon aus, zumindest kein Patent verletzt zu haben, welches diese Bezeichnung verdient.

Schließlich möchte ich noch darauf hinweisen, dass keinerlei Gewährleistung für die hier veröffentlichten Quelltexte besteht, da diese kostenlos lizensiert werden. Der hier veröffentlichte Quelltext steht so zur Verfügung, "wie er ist", ohne irgendeine Gewährleistung, weder ausdrücklich noch implizit, einschließlich – aber nicht begrenzt auf – Marktreife oder Verwendbarkeit für einen bestimmten Zweck. Das volle Risiko bezüglich Qualität und Leistungsfähigkeit der Programme bzw. des vorgestellten Mikroprozessors liegt bei Ihnen. Sollte sich eines der Programme oder der Mikroprozessor als fehlerhaft herausstellen, liegen die Kosten für notwendigen Service, Reparatur oder Korrektur bei Ihnen. In keinem Fall ist irgendein Copyright-Inhaber oder irgendein Dritter, der diese Programme oder die Hardware-Beschreibungen gemäß der GNU GPL modifiziert oder verbreitet hat, Ihnen gegenüber für irgendwelche Schäden haftbar, einschließlich jeglicher allgemeiner oder spezieller Schäden, Schäden durch Seiteneffekte oder Folgeschäden, die aus der Benutzung der Programme oder der Hardware-Beschreibungen oder der Unbenutzbarkeit derselben folgen (einschließlich – aber nicht beschränkt auf – Datenverluste, fehlerhafte Verarbeitung von Daten, Verluste, die von Ihnen oder anderen getragen werden müssen, oder dem Unvermögen des Programms, mit irgendeinem anderen Programm zusammenzuarbeiten), selbst wenn ein Copyright-Inhaber oder Dritter über die Möglichkeit solcher Schäden unterrichtet worden war.

Inhalt

1 Einleitung und Übersicht

Wie viele andere Techniker und Ingenieure bastelte ich bereits in meiner Jugend mit analoger und digitaler Hardware. Damals natürlich im Wesentlichen mit einzelnen Operationsverstärkern, Gattern und Flipflops. Als mein Vater 1990 seinen ersten PC erwarb, begann ich mit dem Programmieren. Während meines Studiums der Elektrotechnik/Regelungstechnik an der TU Darmstadt wurde aus dem Basteln zunehmend ein Entwickeln, auch Mikrocontroller und Peripheriebausteine kamen zum Einsatz. Um im Rahmen meiner ersten Anstellung echtzeitfähige Software zur Modellierung und Regelung von verfahrenstechnischen Prozessen erstellen zu können, benötigte ich schließlich noch eine Portion numerische Mathematik, die aus heutiger Sicht im Ingenieur-Studium viel zu kurz kommt. Heute beschäftige ich mich mit der Integration von Sicherheitssystemen auf Schienenfahrzeugen, habe also mit der eigentlichen Hard- oder Softwareentwicklung nicht mehr viel zu tun. Dennoch ist es vorteilhaft, wenn man die Technik der Lieferanten etwas genauer versteht. Programmierbare Logik war eine der wenigen mir bis dahin völlig unbekannten Welten, so beschloss ich, dies zu ändern.

Nach ein paar Tagen allgemeiner Orientierung (was ist ein PLA, ein CPLD oder ein FPGA, welche Software braucht man etc.) bestellte ich den Spartan3-Starter Kit von Digilent als wohl eine der günstigsten Variaten, in die Welt der FPGAs einzutreten. Die ersten Schaltungen waren in VHDL schnell beschrieben und so sollte als nächstes ein digitaler Funktionsgenerator mit zwei Kanälen, 4-zeiliger LCD-Anzeige etc. entstehen. Soweit kein Problem, allerdings fiel auf, dass der Funktionsgenerator an sich nur einen Bruchteil eines FPGAs benötigt, die für die Bedienung erforderliche Hardware im FPGA allerdings in der Größenordnung von mehreren 100 Slices zuzüglich RAM liegt und der Bedienkomfort trotzdem nicht das ist, was man von dergleichen Systemen erwartet. Die geplante Erweiterung um einen USB-Anschluss und einen A/D-Wandler war auf dem Niveau jedenfalls nicht zu realisieren. Einzige Lösung war ein Mikroprozessor. Selbstverständlich im FPGA - dass dies machbar wäre, war mir aus diversen Artikeln und Foren-Beiträgen klar. Warum ich keinen fertigen Prozessor wie den Picoblaze eingebunden habe? Ich wollte doch was lernen - und dafür ist der Weg bekanntlich das Ziel.

1.1 Überlegungen zur Architektur

- Da die wirklich kritischen Operationen ja mit spezieller separater Hardware im FPGA erledigt werden können, wird für die allermeisten Anwendungen ein Prozessor der nach heutigen Maßstäben unteren bis mittleren Leistungsklasse ausreichen. Wer einen Hochleistungs-Prozessor benötigt, der wird auf separate Prozessoren oder DSPs oder auf FPGAs mit eingebautem Prozessor zurückgreifen. Zudem bin ich sicher nicht in der Lage, mal eben in ein paar Wochen einen High-End-Prozessor zu entwickeln.
- Die meisten Messwerterfassungen benötigen 10-12 Bit, will man noch ein bisschen skalieren, läuft es auf 16-bit Rechnungen hinaus. Audio-Anwendungen benötigen ebenfalls 16 Bit Datenbreite pro Kanal.
- Die Spartan3-Architektur (SLICEM) legt nahe, eine Registerbank aus 16 Registern aufzubauen, wobei zwei Register gleichzeitig gelesen werden können.
- Befehle sollten immer dieselbe Breite haben, aber andererseits den wertvollen Speicher im FPGA auch nicht verschwenden. 16 Bit könnten reichen, um den Befehl und bis zu drei Operanden mit jeweils 4 Bit zu codieren.
- Ein modernes, kleines FPGA wie der Spartan3-200 hat schon 12x4 kB Speicher. Der Adressbereich sollte also mindestens 64 kB umfassen.
- Die Addressierungsarten unmittelbar, direkt, indirekt und indirekt mit Basis sollen unterstützt werden. Diese Adressierungsarten sind sicher die grundlegenden und werden wohl von jedem Assembler-Programmierer und von jedem C-Compiler vorausgesetzt.
- Die genannten Punkte sprechen eindeutig für eine Datenbus-Breite von 16 Bit. Alle Operationen sollen sich auf 16-Bit Werte beziehen. Lediglich das Schreiben in RAM soll auch mit 8-Bit Breite möglich sein.
- Speicher gibt es im FPGA quasi beliebig skalierbar, sich selbst verändernder Code ist im Embedded-Bereich nicht üblich, somit spricht alles für die schnellere und einfachere Harvard-Architektur.
- Es sollte möglich sein, ein größeres externes Daten-RAM oder Daten-ROM einfach ansprechen zu können, also sollten entsprechende spezielle Befehle vorhanden sein.
- Da im Bereich der Messwerterfassung oder Regelung häufig Multiplikationen erforderlich sind, soll der Prozessor über Multiplikationsbefehle verfügen. Ein modernes FPGA bietet schnelle Multiplizierer - also optimale Voraussetzungen.
- Divisionen sind erheblich seltener als Multiplikationen. Nur wenige Algorithmen benötigen sie überhaupt. Bei der Umrechnung von Zahlen in Dezimal-Darstellung, wie sie für die Anzeige häufig benötigt wird, ist eine wiederholte Subtraktion von Zehnerpotenzen im Allgemeinen schnell genug (durchschnittlich etwa 20-25 Takte pro Dezimalstelle bei geschwindigkeitsoptimierter Programmierung

auf dem ssfp16). Andererseits benötigt eine mittels Additionen und Subtraktionen programmierte 16-Bit Division für beliebige Divisoren grob überschlagen 100-200 Takte, berücksichtigt man auch das ggf. notwendige Freimachen von Registern, Funktionsaufrufe etc. Von daher wurden Befehle für die Division vorzeichenloser und vorzeichenbehafteter Zahlen vorgesehen. Exemplarisch wurde ein relativ kleiner Mehrschritt-Dividierer implementiert, welcher 17 Takte benötigt. Damit ist er für die Umrechnung in Dezimal-Darstellung nur unwesentlich schneller als eine wiederholte Subtraktion.

- Um Peripherie effizient verwalten zu können, sollen separate I/O-Befehle sowie ein Interrupt-Eingang vorhanden sein.
- Aus Zuverlässigkeitsgründen sollte eine Parity-Prüfung für Daten und Befehle erfolgen.

1.2 Ergebnis

Mit diesen Rahmenbedingungen wurde in mehreren Iterationsschritten ein auf den Spartan3 zugeschnittener 16-bit Prozessor entwickelt. Durch sehr effizienten Einsatz der zur Verfügung stehenden Komponenten des Spartan3 und gleichzeitige Optimierung der Codierung konnte ein Flächenbedarf von nur etwa 200 Slices und gleichzeitig eine Taktfrequenz von rund 66 MHz auf der langsamen Variante (Geschwindigkeitscode -4) erreicht werden. Hierbei wird für alle Befehle mit Ausnahme des Ladens aus dem Speicher und der Division grundsätzlich ein Takt benötigt. Insgesamt ergibt sich damit ein im Vergleich zu "Nachbauten" existierender Standardprozessoren wesentlich besseres Verhältnis von Geschwindigkeit zu Fläche.

Gleichzeitig muss natürlich darauf hingewiesen werden, dass die genannten Eckdaten wahrscheinlich nur mit FPGAs der Spartan3 und VirtexII-Familien erreicht werden können bzw. es nicht möglich war, alle VHDL-Beschreibungen Hardware-unabhängig zu halten. Zumindest das WebPack 8.1 ist an einigen Stellen nicht in der Lage, die Hardware aus einer Verhaltensbeschreibung heraus ideal auszunutzen.

Will man nicht Befehle von Hand codieren, muss ein Assembler her. Da ich auf die Schnelle nichts Brauchbares gefunden habe (entweder ohne jegliche Dokumentation oder zu einfach oder zu kompliziert), habe ich einen relativ einfacher Assembler in C geschrieben. Dieser ist in Kapitel 3 dieser Dokumentation beschrieben. Er ist hoffentlich mit jedem existierenden C-Compiler zu kompilieren, egal auf welcher Maschine. Ich nutzte sowohl gcc unter Linux als auch Dev-C++ mit mingw32.

Da sich in einem FPGA relativ schwer Fehler suchen lassen und ich mit der VHDL-Simulation des Prozessors wahrscheinlich heute noch am simulieren wäre, schrieb ich außerdem einen Simulator in Java, welcher gleichzeitig als Debugger dient. Der Prozessor wurde dabei nahezu auf RTL-Ebene abgebildet, was die Suche von Fehlern im VHDL-Code vereinfacht, aber

natürlich für das Debuggen von Assembler-Programmen unnötig langsam ist. Insgesamt ist der Simulator wirklich extrem einfach gehalten, aber dafür hat er auch eine sehr einfach zu bedienende Oberfläche. Letztlich benötigt habe ich ihn dann doch nur für das Debuggen von Assembler-Programmen, da im VHDL-Code von Anfang an wider Erwarten keine wesentlichen Fehler waren.

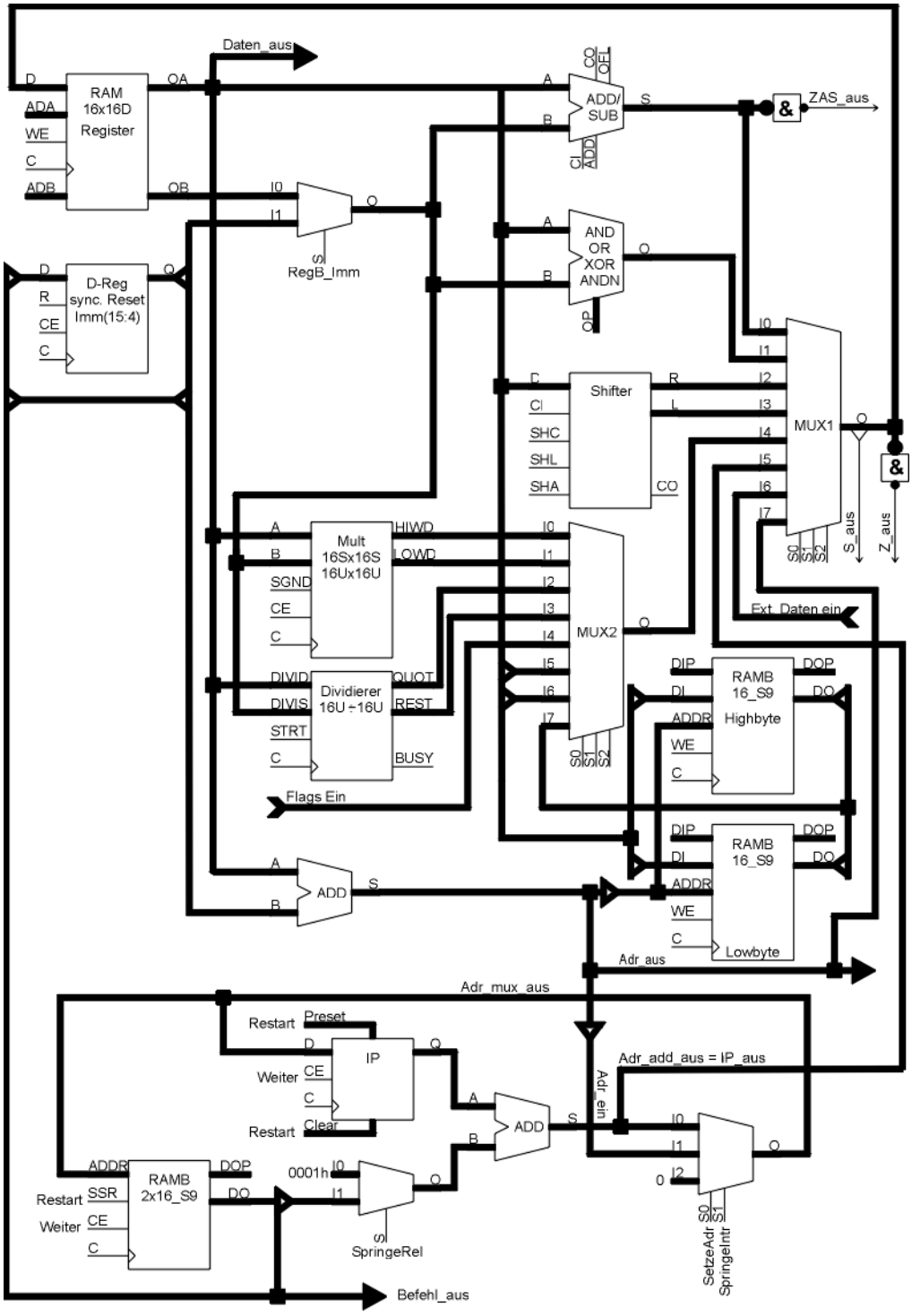
Was vielleicht noch fehlt, ist ein C-Compiler. Da dieser für das Testen des Prozessors nicht nötig ist, habe ich bislang darauf verzichtet, mich auch noch in anpassbare C-Compiler einzuarbeiten. Da gibt es sicherlich Spezialisten, die das wesentlich besser und schneller hinbekommen als ich. Aufgrund des doch begrenzten Befehlsspeichers sowie in Anbetracht der Anwendungen von FPGA-Prozessoren überhaupt ist ein Compiler aus meiner Sicht doch eher "nice to have" als unbedingt notwendig.

2 Prozessor

Der Prozessor-Kern ist in der Datei "ssfp16_test.vhd" beschrieben. Das angehängte "_test" bedeutet dabei nur, dass einige Signale zusätzlich herausgeführt sind - auf Größe und Geschwindigkeit hat dies keinen Einfluss.

Der Befehlssatz ist in ssfp16_befehlssatz.pdf tabellarisch dargestellt.

Der ssfp16 ist ein Prozessor in Harvard-Architektur, d. h. Befehle und Daten benutzen zwei völlig unabhängige Speicherbereiche mit unabhängiger Adressierung. Die Busstruktur des ssfp16-Kerns ist in Bild 1 dargestellt. Der obere Teil stellt den Datenpfad dar, im unteren Teil ist der Befehlspfad zu sehen. Sämtliche interne und externe Steuersignale inklusive der Parity-Bits sind hierin nur angedeutet.



 <>

Bild 1: Busstruktur mit Daten- und Befehlspfad.

2.1 Überblick Datenpfad

Der Datenpfad besteht aus den allgemeinen Registern, dem Konstanten-Register (*imm*), den Einheiten zur Bearbeitung von arithmetischen und logischen Operationen, dem internen Datenspeicher, dem Adress-Addierer sowie einigen Multiplexern zur Auswahl der Operanden und der Ergebnisse. Die meisten der Komponenten sind als einzelne VHDL-Beschreibungen

implementiert, welche struktural zum Datenpfad zusammengeschaltet werden. Die Beschreibungen der untersten Ebene sind zum Teil speziell auf XILINX FPGAs zugeschnitten, da der Synthetisierer (zumindest ISE Webpack 8.1) leider bei Verhaltensbeschreibungen nicht immer optimale Ergebnisse liefert.

2.1.1 Register

Es gibt 16 weitgehend gleichberechtigte 16-bit Register, *r0* bis *r15* benannt. Diese werden als Registerbank bezeichnet. Einige der Register haben jedoch eine spezielle Bedeutung:

- *r0* muss immer den Wert 0 enthalten. Der Assembler verbietet daher jeden schreibenden Zugriff auf dieses Register. Prozessormäßig erfährt dieses Register keinerlei Sonderbehandlung.
- *r13* sollte als Stack-Pointer benutzt werden, wenn ein Stack benötigt wird. Dieses ist jedoch eine rein vom Benutzer abhängige Entscheidung, die weder vom Prozessor noch vom Assembler vorgegeben ist.
- *r14* ist im Prozessor als Speicherort der Rücksprung-Adresse bei Interrupts definiert. Werden keine Interrupts benutzt, kann das Register frei verwendet werden.
- *r15* ist vom Assembler als Speicherort der Rücksprung-Adresse bei call-Befehlen definiert. Seitens des Prozessors könnten genauso gut die Register *r8* bis *r13* für diesen Zweck verwendet werden.

Die Registerbank ist in Form von sogenanntem "Verteilten RAM" im FPGA realisiert, d. h. SLICEMs werden genutzt. Da in vielen Befehlen gleichzeitig zwei Register benötigt werden, ist die Registerbank als Dual-Port-RAM ausgeführt. Der Schreib-Zugriff erfolgt synchron bei gesetztem *WE*, der Lese-Zugriff auf beiden Ports asynchron. Geschrieben werden kann nur auf Port A.

Zusätzlich gibt es einige separate Spezialregister:

- *ip* - Der Befehlszeiger (Instruction Pointer) ist in Form gewöhnlicher D-Flipflops realisiert. Er kann nicht direkt gelesen werden. Wird der Inhalt ausnahmsweise im Programm benötigt, hilft ein *call* auf die nächste Zeile (ohne *ret*). Das Schreiben geschieht mittels *jmp* oder *call*.
- *flags* - Derzeit sind fünf Flags definiert (ebenfalls in Form von D-Flipflops):
  Bit 0 = Übertrag (bei *sub*, *sbb* und *cmp* invertiert), Bit 1 = Überlauf, Bit 2 = Null, Bit 3 = Vorzeichen, Bit 8 = Interrupts zulassen.
  Flags können mit *movs flags* direkt gelesen und geschrieben werden.
- *multh* - Die Bits 31 bis 16 des Ergebnisses einer Multiplikation, bei vorangegangenem *imul* in Zweierkomplement-Darstellung.
- *multl* - Die Bits 15 bis 0 des Ergebnisses einer Multiplikation, bei vorangegangenem *imul* in Zweierkomplement-Darstellung. *multh*

und *multl* sind physikalisch das Ausgangsregister in der speziellen Multipliziereinheit des Spartan3. Sie können daher nicht explizit gesetzt werden.
- *quot* - Enthält den 16-bit Quotienten einer Division. Es besteht aus D-Flipflops.
- *rest* - Enthält den 16-bit Rest einer Division. Es besteht aus D-Flipflops.

Alle Spezialregister sind 16 Bit breit.

2.1.2 Datenfluss

Bei allen arithmetischen und logischen Operationen einschließlich Schieben, Multiplizieren und Dividieren bildet ein Register der Registerbank den ersten Operanden. Dieses steht an Ausgang *OA* an. Wenn ein zweiter Operand benötigt wird, so ist dieser entweder ein zweites Register der Registerbank (Ausgang *OB*) oder ein im Befehl codierter unmittelbarer Wert. Da in einem 16-bit Befehl kein 16-bit Wert Platz hat (schließlich muss der Befehl selbst ja auch noch codiert werden), wird ein unmittelbarer Wert aufgeteilt. Die unteren 4 Bit stehen dabei immer direkt in dem Befehl, in dem sie benötigt werden. Wenn mehr als 4 Bit benötigt werden, müssen die oberen 12 Bit in einem vorgestellten Befehl codiert werden. Diese werden dann einen Takt zuvor im *Imm*-Register gespeichert. Sofort nach dem anschließenden Befehl wird dieses Register wieder zurückgesetzt. Der Befehl zum Setzen des *Imm*-Registers wird ausschließlich vom Assembler selbst generiert und kann nicht manuell eingefügt werden.

Nach der Gatter-Durchlaufzeit der einzelnen Einheiten steht das Ergebnis der gewünschten Operation am Ausgang O des Haupt-Multiplexer *MUX1* an. Mit der nächsten steigenden Taktflanke wird es in das durch *ADA* adressierte Register übernommen, der erste Operand wird also mit dem Ergebnis überschrieben.

Ein Multiplexer ist notwendig, da die Spartan3-FPGAs intern nicht über Three-State-Logik verfügen. Des Weiteren kann mit einer Gatterebene nur ein 8-zu-1 Multiplexer aufgebaut werden, benötigt werden aber mind. 14 Eingänge (ohne *bitswap*). Um zeitkritische Pfade nicht unnötig zu verlängern, wurden zwei 8-zu-1 Multiplexer kaskadiert anstatt einen 16-zu-1 Multiplexer zu beschreiben.

Soll ein Wert in ein RAM oder in eine Peripherie-Einheit kopiert werden, so wird dieser grundsätzlich aus dem durch *ADA* adressierten Register genommen.

Die Adress-Berechnung für RAM und Peripherie ist ebenfalls Teil des Datenpfades. Hierfür wird immer das durch *ADB* adressierte Register zu dem unmittelbaren Wert addiert. Der so berechnete (Adress-)Wert kann in das durch *ADA* adressierte Register geschrieben werden (siehe auch LEA und MOV weiter unten).

2.1.3 Addierer/Subtrahierer

Die arithmetischen Operationen sind:

- *add*
- *sub*
- *adc*
- *sbb*
- *cmp*

Diese Operationen werden vom Addierer/Subtrahierer durchgeführt. Der Spartan3 unterstützt zwar Addierer/Subtrahierer in der benötigten Form hardwaremäßig ideal, leider steht aber keine entsichende Vorlage zur Verfügung - weder zur Instanziierung noch zum Inferieren. Daher habe ich diese Einheit gemäß der XILINX-Unterlagen struktural beschrieben. Eine funktionale Beschreibung wäre im Hinblick auf die Plattform-Unabhängigkeit natürlich schöner, aber seltsamerweise hatte ich an der Stelle keinen Ehrgeiz...

Hinweis: Bei Subtraktionen (einschließlich *cmp*) ist die Bedeutung des Carry-Flags invertiert. D. h. wenn das Ergebnis korrekt ist, ist Carry gesetzt. Wenn also bei *sbb* das Carry-Flag NICHT gesetzt ist, wird zusätzlich 1 subtrahiert. Dieses Verhalten ist von der Spartan3-Hardware so vorgegeben, stört aber meines Erachtens auch nicht, solange man nicht Assembler-Programme mit *jc*-Befehlen portieren will. Dann ist wirklich äußerste Vorsicht angesagt!

2.1.4 Logik-Einheit

Die logischen Operationen sind:

- *and*
- *or*
- *xor*
- *andn*
- *test*

Die logischen Operationen werden in einem Spartan3-Funktionsblock pro Bit abgearbeitet, welcher ja genau die vier benötigten Eingänge bietet.

2.1.5 Schiebe-Einheit

Es existieren fünf Schiebe-Befehle:

- *srl* - logisches Rechtsschieben (0 wird nachgeschoben)
- *sra* - arithmetisches Rechtsschieben (MSB wird nachgeschoben)
- *src* - Rechtsschieben mit Carry (Carry-Flag wird nachgeschoben)
- *shl* - Linksschieben (0 wird nachgeschoben)
- *slc* - Linksschieben mit Carry (Carry-Flag wird nachgeschoben)

Alle Befehle verschieben den Ausgang *OA* der Registerbank um eine Stelle. Es wird immer sowohl nach rechts als auch nach links verschoben, die Auswahl der gewünschten Operation findet erst anschließend durch den Multiplexer *MUX1* statt.

2.1.6 Multiplizierer und Dividierer

Grundsätzlich steht eine vorzeichenlose (*mul*) und eine vorzeichenbehaftete (*imul*) Multiplikation zur Verfügung. Standardmäßig ist nur eine vorzeichenlose Division implementiert, Befehle für vorzeichenbehaftete Division sind jedoch reserviert. Als Operatoren stehen dieselben Quellen wie für die übrigen arithmetischen und logischen Befehle zur Verfügung.

Sowohl Multiplizierer als auch Dividierer besitzen jeweils zwei eigene 16-bit Ausgangsregister, da die Ergebnisse 32 Bit breit sind (Multiplizierer: High-Word und Low-Word des Ergebnisses, Dividierer: Quotient und Rest). Diese Register werden nur durch die jeweiligen Befehle (*mul/imul*, *div/idiv*) verändert, die Ergebnisse von Multiplikationen und Divisionen stehen also bis zur nächsten Multiplikation oder Division zur Verfügung. Mittels des *movs*-Befehls können die Ergebnisse in den internen Registern in die Registerbank übernommen werden.

2.1.7 BITSWAP, BYTESWAP und SETF

- *bitswap* - Alle Bits werden vertauscht (aus MSB wird LSB etc.). Diese Operation wird vom Multiplexer *MUX2* durchgeführt. Zugegebenermaßen eine recht exotische Funktion, die aber im Embedded-Bereich sehr selten mal benötigt wird und softwaremäßig sehr aufwändig ist. Falls aufgrund irgendeiner Erweiterung einmal ein weiterer MUX-Eingang notwendig wäre, wäre dieser Befehl natürlich ein Streich-Kandidat.
- *byteswap* - High-Byte und Low-Byte werden vertauscht. Diese Operation wird ebenfalls vom *MUX2* durchgeführt und wird aufgrund der nicht vorhandenen Byte-Befehle sicher gelegentlich benötigt.
- *setf* - Alle Flags werden gesetzt.

Alle drei Befehle arbeiten mit Ausgang *OA* der Registerbank als Operator.

2.1.8 Adressierung, LEA und MOV

Gemäß Spezifikation in der Einleitung soll es möglich sein, Register mit einer Konstanten zu laden (unmittelbare Adressierung), Werte von einem Register in ein anderes Register zu kopieren, eine konstante Speicherstelle in ein Register zu kopieren oder umgekehrt (direkte Adressierung), eine durch ein Register adressierte Speicherstelle in ein Register zu kopieren oder umgekehrt (indirekte Adressierung) sowie eine durch ein Register plus eine Konstante adressierte Speicherstelle in ein Register zu kopieren und umgekehrt (indirekte Adressierung mit Basisadresse). Zunächst

könnte man auf die naheliegende Idee kommen, die Additionseinheit zur Berechnung der Adresse zu benutzen (also keinen Adress-Addierer einzuführen). Doch dann wäre zumindest bei der in Bild 1 gezeigten Busstruktur das Zielregister gleichzeitig das Adressregister, was in vielen Fällen unpraktisch wäre. Außerdem wäre es unmöglich, direkt eine Konstante zu laden, da diese ja immer zu einem Register addiert würde. Man müsste also vorher das Register immer erst löschen (z. B. durch ein *and rx, 0*), was einen zusätzlichen Befehl im (beschränkten) ROM und einen zusätzlichen Takt bedeuten würde. Da es auch möglich sein soll, ein Register in ein anderes zu kopieren, müsste man zusätzlich eine direkte Verbindung vom Ausgang *OB* der Registerbank zum Multiplexer erstellen. Das Laden einer Konstanten ist aber eine sehr häufig benötigte Operation, daher ist ein bisschen mehr Logik angebracht - der separate Adress-Addierer, welcher grundsätzlich Ausgang *OB* der Registerbank zu einer Konstanten addiert. Er ist 16 Bit breit. Führt man das Ergebnis in ein (beliebiges) Register der Registerbank, so bekommt man den *lea ry, rx+const*-Befehl. Man kann jetzt aber auch Register *r0* (also 0) zu einer Konstanten addieren und in einem beliebigen anderen Register speichern - der *mov rx, const*-Befehl, oder aber ein beliebiges Register zu 0 addieren und in einem andern Register speichern - der *mov ry, rx*-Befehl.

Da nicht die Arithmetische Einheit für die Addition benutzt wird, bleiben Flags hiervon unberührt! (Man könnte natürlich unter Verwendung der ohnehin benötigten Hardware zumindest Vorzeichen- und Null-Flag setzen, das ist aber nicht üblich.)

2.1.9 Speicherbereiche

Im FPGA stehen RAM-Blöcke mit jeweils 4kB zuzüglich Parity-Bits zur Verfügung (BRAM). Diese werden sowohl für den Befehlsspeicher als auch für den Datenspeicher genutzt. Die Wortbreite der BRAMs ist beliebig wählbar aus 1, 2, 4, 8/9, 16/18 und 32/36 Bit. Benutzt man anstelle der 2 BRAMs also 4 BRAMs mit jeweils 4 Bit, so kann die Speichertiefe ohne weitere Änderung verdoppelt werden bis hin zu 32 kB (16 BRAMs je 1 Bit breit). Will man allerdings eine Parity-Prüfung machen, fallen ein bzw. zwei weitere BRAMs an. Alternativ könnte man natürlich auch immer 9-Bit breite BRAMs benutzen und einen externen Multiplexer gemäß des/der höchstwertigen Adressbits nachschalten und so die Parity-Bits der BRAMs auch bei größeren Speichern weiternutzen. Damit ergeben sich je nach Anwendungszweck und geforderter Zuverlässigkeit jedoch viele unterschiedliche Architekturen und so sah ich davon ab, die Speichertiefe bei RAM und ROM generisch zu beschrieben. Ohnehin müsste zumindest die Datei "ssfp16.bmm" manuell geändert werden.

Während für das ROM für sehr einfache Anwendungen ein BRAM (=1024 Befehle) ausreicht, müssen für den Datenspeicher mindestens zwei BRAMs benutzt werden, um High-Byte und Low-Byte unabhängig voneinander schreiben zu können.

Üblicherweise besitzen Prozessoren mit mehr als 8-bit Datenbreite Operationen zur Manipulation einzelner Bytes. Wie man jedoch am Befehlssatz erkennen kann, sind die 16 Bits der Befehle weitgehend ausgereizt. Will man also alle Befehle auch auf Bytes beziehen, so wäre man gezwungen, auch die Befehlsbreite zu vergrößern. Nun könnte man natürlich eines der beiden Parity-Bits des ROM noch als Befehlsbit nutzen, aber das wäre dann schon ein ziemlich unorthodoxes Vorgehen und auch wahrscheinlich schwer auf andere Architekturen übertragbar. Andererseits gibt es nur sehr wenige Punkte, an denen ein 16-Bit Wert anstelle eines 8-Bit Werts von Nachteil ist. Der schreibende Zugriff auf RAM dürfte eine dieser wenigen Anwendungen sein, müsste doch beim Schreiben eines Einzelbytes erst ein Wort gelesen, das jeweils zu überschreibende Byte mittels Verundung auf 0 gesetzt und anschließend durch Veroderung auf den neuen Wert gesetzt werden. Im schlimmsten Fall könnte das eigentlich nicht zu verändernde Byte sogar mittlerweile durch einen Interrupt verändert worden sein und würde dann wieder mit dem alten Wert überschrieben. Folglich müssten die Speicherbereiche sehr genau betrachtet oder grundsätzlich während solcher Operationen Interrupts vom Programm gesperrt werden. Aufgrund dieser Überlegungen entschied ich, prinzipiell keine Byte-Befehle vorzusehen außer zum Schreiben in RAM.

Datenspeicher wird auch bei vielen einfachen Anwendungen in relativ großen Mengen benötigt. Hier werden die vorhandenen BRAMs häufig nicht ausreichen, schließlich wird niemand einen teuren Riesen-FPGA kaufen, nur um ein paar zusätzliche kB RAM zu bekommen. Externer Datenspeicher wird daher häufig eingesetzt werden - sei es in Form eines echten RAM oder in Form eines (Flash-)ROM. Prinzipiell fallen mir drei Möglichkeiten ein, auf externen Speicher zuzugreifen:

1. Der externe Speicher wird als peripheres Gerät angesehen, dem ein bestimmter (großer) Adress-Bereich zugeordnet ist. Dieses Verfahren wird bei Mikrocontrollern gelegentlich angewandt, bringt aber gelegentlich Nachteile.
2. Der externe Speicher wird gleichberechtigt zum internen Speicher mit separaten Befehlen und separatem Adressbereich angesprochen. Dies ist das Standard-Verfahren für Mikroprozessoren ohne internen Speicher. Nachteil ist, dass zusätzliche Befehle nötig sind.
3. Mittels eines DMA-Controllers, welcher parallel zum Prozessor in den internen Datenspeicher schreibt - welcher dann zumindest teilweise als Cache fungieren würde.

Ich habe mich für die zweite Variante entschieden, wobei die dritte Variante aufgrund der Dual-Port-Fähigkeit der BRAMs relativ leicht zusätzlich zu integrieren wäre.

Die Adress-Berechnung erfolgt wie oben beschrieben grundsätzlich mit 16 Bit. Somit können 64 kB Daten bzw. 65536 Befehle direkt angesprochen werden. Mittels eines Adress-Basis-Registers im I/O-Bereich können dennoch leicht auch beliebig große Datenspeicher angesprochen werden.

Insgesamt stehen damit folgende Befehle für das Lesen und Schreiben von RAM zur Verfügung:

- *ld* - Wort aus internem Speicher in Register laden.
- *ldx* - Wort aus externem Speicher in Register laden.
- *st* - Wort von Register in internen Speicher schreiben.
- *stb* - Byte von Register in internen Speicher schreiben.
- *stx* - Wort von Register in externen Speicher schreiben.
- *stxb* - Byte von Register in externen Speicher schreiben.

Der *ld*-Befehl benötigt zwei Takte, da die BRAMs einen Register-Ausgang besitzen, welcher mit der nächsten Taktflanke erst gesetzt wird. Es wäre zwar möglich, durch Versatz der Taktflanken den BRAM-Ausgang im selben Prozessortakt zu setzen, jedoch müsste hierfür die Taktfrequenz deutlich reduziert werden - was insgesamt kontraproduktiv ist.

Hinweis: Bei Wort-Befehlen wird das LSB der Adresse ignoriert. Lesen und Schreiben von Worten erfolgt also immer mit Wort-Ausrichtung.

Hinweis: Bei *stb* und *stxb* und LSB der Adresse = 0 werden die Bits 15 bis 8 des Registers geschrieben, bei LSB = 1 die Bits 7 bis 0!

2.2 Befehlspfad

2.2.1 Befehls-ROM

Die Größe des im FPGA in Form von BRAMs verfügbaren ROM liegt in der für Mikrocontroller typischen Größenordnung von 4-32 kB und sollte für die meisten Anwendungen ausreichen. Andernfalls müsste der Befehlspfad um externes ROM ergänzt werden, was jedoch unter Beibehaltung der restlichen Architektur zu einer deutlichen Geschwindigkeits-Einbuße führen würde. In diesem Fall müsste also zumindest ein Befehls-Pipelining eingeführt werden, um einen ganzen Takt für das Auslesen des ROM zur Verfügung zu haben. Das ROM ist nur vom Befehlszeiger *ip* adressierbar und nur wortweise. Eine ROM-Adresse entspricht also 16 Bit (bzw. 18 Bit mit Parity) und einem Befehl. Von der Adresse-Einheit adressierbar sind somit 65536 Befehle oder 128 kB Befehls-ROM.

Ist *Weiter* gesetzt, so wird mit der nächsten Taktflanke der Befehl an der anliegenden Adresse ausgegeben und im selben Takt abgearbeitet (bzw. bei *ld* und *div* mit der Abarbeitung begonnen).

2.2.2 Befehlszeiger *ip* und Sprünge

Dieses Register ist so breit wie für die Adressierung des ROM benötigt - bei 4 kB = 2048 Befehlen also 11 Bit. Außer bei Start und Neustart enthält es immer die Adresse des gerade am Ausgang des ROM stehenden Befehls, nicht etwa die des nächsten Befehls.

Sofern der am Ausgang des ROM anstehende Befehl kein Sprungbefehl ist und kein Interrupt ausgeführt werden soll, wird der Wert des *ip* vom Adress-Addierer um 1 erhöht und an den Eingang des *ip* und den Adress-Eingang des ROM angelegt. Ist es ein bedingter Sprungbefehl (*jxx* abgekürzt, wobei xx für ein Bedingung steht), wird *SpringeRel* gesetzt und damit statt 1 ein im Befehl codierter vorzeichenbehafteter 8-bit Wert addiert. Dieser muss selbstverständlich vor der Addition vorzeichenbehaftet auf die Breite des *ip* erweitert werden. Somit sind bedingte Sprünge bis 128 Befehle zurück und 127 Befehle nach vorn möglich.

Unbedingte Sprünge (*jmp*) und Funktionsaufrufe (*call*) sind für den Befehlspfad identisch. In jedem Fall wird vom Adress-Addierer im Befehlspfad durch Addition eines Registers und einer Konstanten die neue Adresse berechnet und aufgrund des gesetzen Signals *SetzeAdr* gleichzeitig an die Eingänge von *ip* und ROM gelegt. Der Unterschied besteht lediglich darin, dass bei *call*, also wenn Bit 7 im Befehl gesetzt ist, noch ein *WE* für die Registerbank gesetzt wird und somit der Wert des *ip* zuzüglich 1 in das im Befehl codierte Register geschrieben wird - die Rücksprungadresse. Vom Assembler wird dieses Register bei einem *call*-Befehl immer auf 15 gesetzt, bei einem *int*-Befehl immer auf 14, für den Prozessor sind alle Register von 8 bis 15 möglich.

2.2.3 Interrupts

Für die meisten Anwendungen unentbehrlich sind Interrupts. Externe Interrupt-Anforderungen werden nur akzeptiert, wenn das "Interrupt Enable"-Flag (IE-Flag) gesetzt ist. Nach Start oder Neustart ist dieses zunächst nicht gesetzt, es muss erst explizit im Programm gesetzt werden. Bei Aufruf eines externen Interrupts wird es automatisch auf 0 gesetzt, beim Rücksprung mittels *iret* wird es wieder auf 1 gesetzt. Die Rücksprung-Adresse wird in Register *r14* gespeichert (vom Prozessor vorgegeben), eine automatische Sicherung von Registern auf dem Stack erfolgt nicht, da es keinen dedizierten Stack gibt. Durch das IE-Flag ist jedoch sichergestellt, dass nach einem Interrupt-Aufruf erst alle Register gesichert werden können, ohne die Möglichkeit zu verwehren, rekursive Interrupts zu ermöglichen. Vor einen Interrupt-Rücksprung müssen Interrupts ggf. explizit gesperrt werden, bevor die letzten Register wie Rücksprungadresse und Flags wieder hergestellt werden - beim Rücksprung selbst werden Interrupts dann automatisch wieder freigegeben.

Im Befehlspfad wird ein Interrupt wie folgt abgearbeitet: Zunächst wird *Weiter* von der Steuerung zurückgenommen und *SpringeIntr* gesetzt. Damit wird ein *Interrupt-Takt* eingefügt - nichts anderes als ein von der Hardware angeordneter *call* nach 0 mit Speicherung der Rücksprungadresse in *r14*. Durch die Rücknahme von *Weiter* bleibt der Ausgang des Adress-Addierers auf dem nächsten auszuführenden Befehl stehen - der Rücksprungadresse. Das Einfügen des *call*-Befehls geschieht,

indem ein Multiplexer in der Steuerung anstatt des Ausgangs des ROM (welcher in diesem Fall den bereits ausgeführten Befehl enthält) einen *call*-Befehl zur weiteren Dekodierung gibt. Das Zielregister *r14* wird ebenfalls von der Steuerung generiert.

Da an Adresse 2 das "richtige" Programm beginnt (die erste Adresse nach Start und Neustart, siehe unten), muss an Adresse 0 oder 1 ein *jmp* zum Beginn der Interrupt-Routine stehen. In den meisten Fällen wird an Adresse 0 ein *imm*-Befehl für die oberen 12 Bit der Sprungadresse stehen.

Ein Interrupt-Takt darf auch bei gesetztem IE-Flag nicht jederzeit eingefügt werden:

- Nicht nach einem *imm*-Befehl, da sonst das *imm*-Register automatisch gelöscht würde und nach dem Rücksprung nicht mehr zur Verfügung stünde.
- Nicht während eines Befehls, der mehrere Takte benötigt (*ld*, *ldx*, *div*), da dieser dann nicht ganz ausgeführt würde.
- Nicht wenn gerade ein unbedingter Sprungbefehl (*jmp* oder *call*) ansteht, da aufgrund der Rücknahme des *Weiter*-Signals der Sprung sonst gar nicht ausgeführt würde.

Hinweis: Die Register MULTL, MULTH, QUOT und REST können nicht explizit geschrieben werden. Sie können daher in Interrupts nur dann genutzt werden, wenn sie im Hauptprogramm nicht genutzt werden oder wenn im Hauptprogramm jeweils für die Zeit, in der das Ergebnis benötigt wird, das IE-Flag auf 0 gesetzt wird. Da in Interrupts eigentlich grundsätzlich keine komplexen Rechnungen durchgeführt werden sollten, rechtfertigt diese Einschränkung meines Erachtens keine zusätzlichen, schreibbaren Register und entsprechende Befehle.

2.2.4 Start und Neustart (Restart)

Beim ersten Start (Initialisierung des FPGA) wird ip auf 1 gesetzt und der ROM-Ausgang (=Befehl) auf 0 gesetzt. Gleiches gilt für einen Neustart (Restart). Der ROM-Ausgang 0 entspricht einem *sub r0, r0*, also einem Befehl, der effektiv nichts tut, da *r0* ohnehin 0 ist. Am Adress-Eingang von ROM und *ip* liegt dann 2, die Adresse des ersten Befehls. Kann nicht sicher ausgeschlossen werden, dass die Interrupt-Behandlung an einer Befehls-Adresse kleiner 16 beginnt, so muss der zweite Befehl im Assembler-Programm ein "unnötiger" Befehl sein.

2.3 Steuerung

Im Modul "Steuerung.vhd" werden alle zur Steuerung des Programm- und Datenflusses benötigten Signale generiert. Auch die Auswertung der Bedingungen für bedingte Sprünge geschieht in diesem Modul (vgl. Tabelle 1). Die wesentlichen Signale für die Steuerung des

Programmablaufs wurden bereits im Abschnitt "Befehlspfad" genannt. Alle anderen dürften selbsterklärend sein.

Tabelle 1: Bedingte Sprungbefehle

Code	Mnemonic	Flags	Bemerkung
0	jbe	not C or Z	Unsigned
1	jae, jc	C	Unsigned
2	jb, jnc	not C	Unsigned
3	ja	not (not C or Z)	Unsigned
4	jle	(S xor O) or Z	Signed
5	jge	not (S xor O)	Signed
6	jl	S xor O	Signed
7	jg	not ((S xor O) or Z)	Signed
8	js	S	
9	jns	not S	
10	jo	O	
11	jno	not O	
14	jz, je	Z	
15	jnz, jne	not Z	

2.4 Paritätsprüfung

Die Berechnung und Auswertung der Paritäts-Bits (Parity-Bits) des Datenspeichers erfolgt im Modul "datenpfad.vhd", die Auswertung für den Befehlsspeicher im Modul "befehlspfad.vhd". Es wurde ungerade Parität gewählt, da so auch leicht geprüft werden kann, ob eine Speicherstelle bereits initialisiert wurde. Wird lesend auf eine nicht initialisierte Speicherstelle zugegriffen, so enthält diese im Paritäts-Bit eine 0 und löst somit einen Paritätsfehler aus. Paritätsfehler werden als Signal nach außen gegeben und können dort beispielsweise eine Neu-Initialisierung, einen Interrupt oder eine Prozessorumschaltung auslösen.

Wird keine Paritätsprüfung benötigt, kann diese in den zuvor genannten Quelldateien entfernt werden und somit ca. 12 LUTs gespart werden. Der Enable-Eingang der Datenspeicher BRAMs kann dann immer auf 1 gesetzt werden.

Hinweis: Aufgrund eines Fehlers im ISE Webpack 8.1 ist es mit dieser Software nicht möglich, die Paritäts-Bits bei Erst-Start und Neustart richtig zu initialisieren. Daher wird beim Start immer ein Paritätsfehler für Befehls- und Datenspeicher ausgelöst. Je nach Behandlung der Paritätsfehler muss dies ggf. berücksichtigt werden.

3 Assembler

In diesem Abschnitt wird zunächst die Assembler-Sprache beschrieben, dann das Assembler-Programm kurz erläutert und schließlich ein paar Hinweise zur Programmierung gegeben.

3.1 Assembler-Sprache

Prinzipiell wird zwischen "Anweisungen" und "Befehlen" unterschieden. Anweisungen sind nur für den Assembler bestimmt, damit dieser weiß, was er zu tun hat. Anweisungen werden vom Entwickler des Assembler-Programms definiert und sind nicht vom Prozessor vorgegeben. Befehle dagegen werden vom Assembler unmittelbar in Maschinencode umgewandelt. Sie sind damit vom Prozessor vorgegeben. Einzig sogenannte "Pseudo-Befehle" können und sollen vom Assembler-Entwickler zusätzlich definiert werden. Dieses sind Befehle, die eigentlich Spezialfälle anderer Befehle sind. Beispiele sind:

- *mov rx, ry = lea rx, ry+0* ,
- *ret = jmp r15* oder
- *iret = jmp r14*

Kommentare beginnen wie üblich mit ";" und enden am Zeilenende.

Bei Anweisungen und Befehlen wird nicht zwischen Groß- und Kleinschreibung unterschieden, jedoch muss das gesamte reservierte Wort groß oder klein geschrieben sein.

Ein Parser zur Berechnung von Ausdrücken zur Assemblierzeit ist nicht implementiert. Wenn eine Konstante erwartet wird, kann diese also nicht erst aus anderen Symbolen berechnet werden.

Eine Marke muss immer am Zeilenanfang stehen.

Im Adress-Teil von Befehlen ist die Verwendung des "+" bzw. "-" optional, es ist also erlaubt, *ld r1, [r2]* anstelle von *ld r1, [r2+0]* zu schreiben oder *stb [27], r1* anstelle von *stb [r0+27], r1*.

3.1.1 Anweisungen

Der ssfp16asm kennt die in Tabelle 1 dargestellten Anweisungen.

Tabelle 1: Anweisungen

	.DATA	
	.EDATA	
	.CODE	
	.ORG	Dezzahl | Hexzahl | Symbol
	.ALIGN	Dezzahl | Hexzahl | Symbol (1, 2, 4, 8,..., 4096)
Marke[:]	*.EQU*	Dezzahl | Hexzahl
[Marke[:]]	*.DB*	Dezzahl | Hexzahl | Symbol | Zeichen | Zeichenkette [,Dezzahl | Hexzahl | Symbol | Zeichen | Zeichenkette]...
[Marke[:]]	*.DW*	Dezzahl | Hexzahl | Symbol | Zeichen | Zeichenkette [,Dezzahl | Hexzahl | Symbol | Zeichen | Zeichenkette]...
[Marke[:]]	*.RESB*	Dezzahl | Hexzahl | Symbol
[Marke[:]]	*.RESW*	Dezzahl | Hexzahl | Symbol

.DB und *.DW* reservieren und initialisieren Bytes oder Words entweder im internen Datenspeicher oder im externen Datenspeicher. **Hinweis:** Zeichen und Zeichenketten können auch mit *.DW* initialisiert werden. In diesem Fall wird ein Wort pro Zeichen geschrieben. Dies ist im Zusammenhang mit Paritätsprüfungen sinnvoll (s. u.)

.RESB und *.RESW* reserviert die durch die folgende Zahl angegebene Anzahl Bytes oder Words entweder im internen Datenspeicher oder im externen Datenspeicher.

Mit den Anweisungen *.DATA*, *.EDATA* und *.CODE* wird dem Assembler mitgeteilt, auf welche Speicherbereiche sich die folgenden Anweisungen beziehen. Befehle dürfen dabei nur nach *.CODE* stehen, Speicherinitialisierungen und -reservierungen nur in *.DATA* oder *.EDATA*.

Mit *.EQU* werden Konstanten definiert. Diese dürfen in allen Bereichen definiert werden.

.ORG setzt den Adresszähler des aktuellen Bereichs auf den angegebenen Wert. Dieser Wert darf nicht kleiner sein, als der aktuelle Wert.

Mit *.ALIGN* wird die Ausrichtung der Daten im Datenspeicher definiert:

- *.ALIGN 1*: Standardausrichtung: Bytes an Bytegrenzen, Words an Wordgrenzen.
- *.ALIGN 2*: Auch Bytes an Wordgrenzen ausrichten.
- *.ALIGN 4, 8,...*: Alle Daten an DWord/QWord/...-Grenzen ausrichten. Wert muss eine Zweierpotenz sein.

Bei mehreren Werten hinter *.DB* oder *.DW* gilt die Ausrichtung nur für den ersten Wert, alle weiteren werden direkt angehängt.

Es können mehrere *.ALIGN*-Anweisungen unmittelbar hintereinander stehen. In diesem Fall wird der Adresszähler gemäß der Anweisung mit der größten Ausrichtung gesetzt, für folgende Anweisungen gilt der letzte Ausrichtungswert. Beispiel:

```
        .align 0x10

        .align 2

var1:   .db    5

text1:  .db    "Das ist Text1!", 0

text2:  .dw    "Das ist Text2!", 0
```

var1 wird an eine durch 0x10 teilbare Adresse geschrieben. *text1* und *text2* werden an Wort-Grenzen begonnen.

Wichtiger Hinweis: Werden bei Nutzung der Paritätsprüfung einzelne Bytes initialisiert oder vom Programm geschrieben, so wird das andere Byte eines Worts nicht als initialisiert gekennzeichnet. Bei lesendem Zugriff auf das Wort wird daher ein Paritätsfehler generiert. Daher muss vom Programmierer sichergestellt werden, dass nur auf vollständig initialisierte Worte lesend zugegriffen wird. Dies kann beispielsweise durch Verzicht auf *.DB* geschehen oder dadurch, dass hinter *.DB* immer eine gerade Anzahl von Werten steht. Bei Verwendung der Assembler-Option "-i" muss entsprechend nach *.RESB* eine gerade Zahl folgen. Ohne Verwendung der Option "-i" muss sichergestellt werden, dass immer erst beide Bytes eines Wortes vom Programm geschrieben wurden, bevor auf dieses Wort lesend zugegriffen wird.

3.1.2 Befehle

Befehle haben immer folgende Syntax:

[Marke[:]] *Befehl* [Operanden]

Befehle mit unmittelbaren Operanden (z. B. *addi*, *sbbi* etc.) werden nur durch die angegebenen Operanden unterschieden, nicht durch den Befehlsnahmen. Anstelle von *addi* kann und muss also *add reg, imm* geschrieben werden.

3.1.3 Operanden

Entsprechend ihrer Operanden lassen sich die Befehle in Gruppen gemäß Tabelle 2 einteilen. Diese Befehlsgruppen werden vom Assembler zur Syntaxprüfung und Zuweisung in die einzelnen Befehlsfelder herangezogen.

Tabelle 2: Befehlsgruppen

Gruppe	**Syntax**	**Befehle**	**Feld-Zuweisungen**
OP_RR_RI	"befehl reg1, reg2" "befehl reg1, wert":	*add, sub, adc, sbb, cmp* *and, or, xor, andn, test* *mul, imul, div, idiv*	reg1 => RegA reg2 => RegB_imm4 wert => RegB_imm4 und ggf. Imm12
OP_R	"befehl reg"	*shl, slc, sra, srl, src, bitswap, byteswap, setf*	reg => RegA
OP_LEA	"*lea* reg1, reg2+wert"	*lea*	reg1 => RegA reg2 => RegB wert => Imm4 und ggf. Imm12
OP_MOV	"*mov* reg1, reg2" "*mov* reg1, wert"	*mov*	reg1 => RegA reg2 => RegB wert => RegB_imm4 und ggf. Imm12
OP_MOVS	"*movs* reg, regspec"	*movs*	reg => RegA regspec = *multh, multl, quot, rest, flags* => Subsubcode

OP_LD	"befehl reg1, [reg2+wert]" "befehl reg1, [reg2]" "befehl reg1, [wert]"	*ld, ldx, in*	reg1 => RegA reg2 => RegB wert => Imm4 und ggf. Imm12
OP_ST	"befehl [reg1+wert], reg2" "befehl reg1, [reg2]" "befehl reg1, [wert]"	*st, stb, stx, stxb, out*	reg1 => RegB reg2 => RegA wert => Imm4 und ggf. Imm12
OP_CALLJMP	"befehl reg+wert" "befehl reg" "befehl wert"	*call, jmp, int*	reg => RegB wert => Imm4 und ggf. Imm12 *call*: 15 => RegA *int*: 14 => RegA *jmp*: 0 => RegA
OP_JXX	"befehl wert"	*ja, jae, jb, jbe,* *jg, jge, jl, jle,* *jz, jnz, je, jne,* *jo, jno, js, jns,* *jc, jnc*	Bedingung => Bedingung wert => disp
NO_OP	"befehl"	*ret, iret*	0 => RegA *iret*: 14 => RegB *ret*: 15 => RegB 0 => Imm4

Ist mit dem Befehl eine schreibende Operation verbunden, so steht das Ziel immer an erster Stelle. Ist mit dem Befehl eine logische oder arithmetische Verknüpfung verbunden, so gilt stets (Ziel :=) Operand1 Verknüpfung Operand2. Beispiele:

```
SUB   R1, R2              ; R1 := R1 - R2
STX   [R1+0x1234], R2     ; externer Speicher [R1+0x1234] := R2
LD    R1, [R2+0x1234]     ; R1 := interner Speicher[R2+0x1234]
```

3.2 Kurzbeschreibung des Assembler-Programms

Der Assembler liest genau eine Eingangsdatei mit Assembler-Quelltext. Gemäß dieser Datei werden erzeugt

- eine Datei mit Endung *.lst*, welche Symboltabellen und die übersetzten Befehlen in Textform enthält,
- eine Datei mit Endung *.mem*, welche die für die Initialisierung des FPGA-internen Datenspeichers und Befehlsspeichers notwendigen Werte, also ein Speicher-Abbild, enthält,
- und eine Datei mit Endung *_edata.mem*, welche ggf. Werte für die Initialisierung von externem nichtflüchtigen Datenspeicher enthält. Sind im Bereich *.EDATA* keine Initialisierungen erfolgt, so ist die Datei leer.

3.2.1 Optionen

Folgende Optionen stehen derzeit zur Verfügung:

- -bp : Paritätsbits für Befehle in 'name.mem' setzen.
- -dp : Paritätsbits für Daten in 'name.mem' setzen.
- -i : Mit .RESB und .RESW reservierten Speicher auf 0 initialisieren. Ist diese Option angegeben, wird auch das Parity-Bit auf 'initialisiert' gesetzt.

3.2.2 Aufbau und Funktionsweise

Das Hauptprogramm des Assemblers befindet sich in der Datei *ssfp16asm.c*. Die lexikalischen Eigenschaften einer Assembler-Quelltext-Datei werden in der Datei *ssfp16asm.lex* beschrieben. Diese Datei ist Eingangsdatei für das Programm "flex", welches daraus einen C-Quelltext *lex.yy.c* generiert. In *lex.yy.c* befindet sich eine Funktion *yylex()*, welche durch die Assembler-Quelltext-Datei geht und bei jedem Erkennen eines der in *ssfp16asm.lex* beschriebenen Elemente eine entsprechende Funktion aufruft. Diese Funktionen wiederum sind in *lex_fun.c* definiert. Bereits in dieser Stufe werden alle Initialisierungen der *.DATA* und *.EDATA*-Bereiche in die entsprechende Memory-Datei geschrieben.

Nach Durchlauf von *yylex()* befinden sich eine Symboltabelle und eine Befehlstabelle im Speicher. Einige der Symbole (die Marken im Code) müssen nun noch aktualisiert werden, da sie entweder beim Durchgehen der Quelltext-Datei erst später definiert wurden oder weil sich durch benötigte *imm*-Befehle Marken verschoben haben. Sind alle Werte der Symbole stabil, können die Befehle mit aktualisierten Werte für unmittelbare Konstanten in die *.mem*-Datei geschrieben werden.

Hinweis: Der Assembler produziert Daten (.mem-Datei), welche von dem im ISE Webpack enthaltenen Programm "data2mem" in die FPGA-Konfigurationsdatei eingefügt werden. Diese Daten enthalten in der

derzeitigen Version grundsätzlich Parity-Bits für interne Befehls- und Datenspeicher (Anzahl der gesetzten Bits incl. Parity ist ungerade). Aufgrund eines Fehlers in der mit ISE Webpack 8.1 gelieferten Version von "data2mem" kann jedoch bei Benutzung der Parity-Bits nur zwei Drittel des internen Adressraums initialisiert werden. Weitere Werte in der .mem-Datei werden als Fehler gewertet. Daher muss derzeit in der (eigentlich korrekten) .mem-Datei manuell mindestens das letzte Drittel der Initialisierungswerte des Daten-RAM mittels "/*" und "*/" auskommentiert werden. Dadurch kann natürlich jeweils ein Drittel des Befehls- und Datenspeichers nicht initialisiert werden.

3.3 Hinweise zur Programmierung

3.3.1 Interrupt-Routinen

Es erfolgt keinerlei automatische Sicherung von Registern. Erste Aufgabe eines Interrupts ist es daher, die Rücksprungadresse aus *r14* auf dem Stack zu speichern, dann die Flags in *r14* zu laden (das einzige Register, welches in diesem Augenblick zur Verfügung steht) und von dort ebenfalls auf den Stack zu legen. Wichtig hierbei ist, dass beim Sichern der Rücksprungadresse auf dem Stack die Flags nicht verändert werden dürfen. Somit darf der Stack-Zeiger noch nicht erniedrigt werden, daher *st [r13-2], r14* benutzen und erst nach Kopieren der Flags in *r14* den Stack-Zeiger um 4 erniedrigen. Beim Beenden des Interrupts entsprechend umgekehrt.

3.3.2 INT-Befehl

Dieser Befehl kann genutzt werden, um Interrupt-Handler vom Programm aus aufzurufen. Das Sprungziel wird dabei in der Regel nicht 0 sein. In wie weit dieser Befehl sinnvoll ist sei dahingestellt. Es ist zu beachten, dass das IE-Flag durch den *INT*-Befehl im Gegensatz zu externen Interrupts NICHT automatisch gelöscht wird. Dies muss daher manuell vor Aufruf des Interrupts getan werden. Vom *iret*-Befehl wird es wie auch bei externen Aufrufen wieder gesetzt.

3.3.3 NOP-Befehl

Ein *nop*-Befehl ist zwar nicht definiert, jedoch hat z. B. *lea rx, rx+0* mit einem beliebigen Register (außer *r0*) genau diese Wirkung.

4 Simulator / Debugger

4.1 Bedienung

Voraussetzung für den Einsatz ist die Verfügbarkeit einer Java-Runtime-Umgebung. Getestet wurde mit SUN j2re-1.4.2 unter Linux und Windows. Der Simulator wird von der Kommandozeile aus mit "java Sim" gestartet.

Derzeit sind keine Möglichkeiten zur Auswahl der Dateien implementiert, daher müssen sich die Dateien "test.lst", "test.mem" und "test_edata.mem" im Verzeichnis der Simulator-Klassen befinden. Wer mag, kann ja mal eine Menüleiste mit typischen Datei-Funktionen programmieren. Die Speicherbereiche für Befehle und Daten in "test.mem" müssen Parity-Informationen enthalten.

Nach dem Start präsentiert sich eine Oberfläche mit drei Bereichen: Links werden Register und Flags angezeigt, in der Mitte der Inhalt der Datei "test.lst" und im rechten Bereich oben der Inhalt des internen Datenspeichers, darunter der des externen Datenspeichers gemäß den Inhalten der Dateien "test.mem" und "test_edata.mem".

In der Mitte wird der als nächstes auszuführende Befehl mittels eines blauen Dreiecks markiert (zur Zeit wird der aktuelle Befehl allerdings nicht automatisch in den sichtbaren Bereich gescrollt). Klickt man mit der Maus links neben eine Zeile, wird ein Haltepunkt für diese Zeile gesetzt, erkennbar an dem roten "Stop"-Zeichen. Durch Drücken von "Ein Schritt" wird genau ein Befehl ausgeführt, durch Drücken von "Bis Haltepunkt" wird die Programmausführung bis zum Erreichen eines Haltepunkts fortgesetzt. Wird eine Zeile angeklickt, erscheint sie grau hinterlegt. Bei Drücken von "Bis Zeile" wird das Programm bis zum Erreichen des markierten Befehls fortgesetzt. Wenn die Taste "Reset" gedrückt wird, wird der Zustand nach Neuinitialisierung des FPGA eingenommen, also wie nach einem Erst-Start. Die Speicher-Dateien werden dabei erneut eingelesen.

Register können im linken Bereich durch Eingabe neuer Werte verändert werden. Wird das IP-Register verändert, äußert sich dies erst nach Fortsetzung des Programms mittels einer der Tasten "Ein Schritt", "Bis Zeile" oder "Bis Haltepunkt". Der im IP-Register gesetzte Wert zeigt dabei auf den als nächstes auszuführenden Befehl. Werte im Datenspeicher können durch Eingabe der Adresse und des neuen Werts ebenfalls wortweise verändert werden. Unter Windows wird der Eingabe-Cursor leider nicht richtig gesetzt und muss daher mittels Maus auf andere Stellen bewegt werden.

Alle seit dem letzten Halt veränderten Register und Speicherstellen werden durch rote Schrift markiert, alle manuell veränderten Register blau.

4.2 Aufbau und Funktionsweise

Von einer detaillierten Beschreibung möchte ich hier absehen, da der Simulator/Debugger sicher kein Musterstück zukunftsweisender Technik ist oder sein will - entsprechend nachlässig ist auch der Programmierstil (an einigen Stellen extrem ineffizient oder unergonomisch).

Der Prozessor wurde nahezu auf RTL-Ebene abgebildet, um die Suche von Fehlern im VHDL-Code zu ermöglichen. Die Dateien "Processor.java",

"Datenpfad.java", "Befehlspfad.java" und "Steuerung.java" entsprechen funktional nahezu exakt den gleichnamigen VHDL-Dateien. Dieses Vorgehen ist natürlich relativ kompliziert, da die zeitliche Parallelität der zwischen den einzelnen Modulen des Prozessors ausgetauschten Signalen berücksichtigt werden muss. Zudem macht diese Struktur das Debuggen von Assembler-Programmen sehr rechenintensiv, was sich bei längeren Programmläufen zwischen Halten durchaus bemerkbar macht.

Für die Simulation von Peripherie wie externem RAM, Interrupt-Controller etc. steht das Modul "IOSimulator.java" zur Verfügung. Derzeit ist nur das externe RAM modelliert.

5 Installation und Tests

5.1 Installation

"ssfp16.zip" in einem beliebigen Verzeichnis entpacken.

5.2 Testumgebung

Um den Prozessor wirklich ausprobieren zu können, ist beispielsweise ein Entwicklungsboard mit einem beliebigen Spartan3 (oder evtl. Virtex II) notwendig. Hier benutze ich das Board von Digilent, welches es für rund 100 Euro zzgl. MwSt. in Deutschland mit kurzen Lieferzeiten zu kaufen gibt. Für dieses Board ist eine kleine Testumgebung realisiert, welche die Anzeige einiger Signale, das schrittweise Abarbeiten des Programms sowie das Auslösen von Interrupts ermöglicht. So kann der Prozessor vollständig getestet werden.

Auf der 4-stelligen 7-Segment-Anzeige werden in Abhängigkeit der Schalter S0 bis S6 die in Tabelle 1 aufgelisteten Signale dargestellt.

Tabelle 1: 7-Segment-Anzeigen

S6-S0	**Digit3**	**Digit2**	**Digit1**	**Digit0**
000 0000 - 000 1111	I/O-Ports 0 bis 15			
001 xxxx	adr_aus			
010 xxxx	Daten_aus			
011 xxxx	regA_adr	flags(3..0)	'0' & mux_wahl1	'0' & mux_wahl2
100 xxxx	'0' & regWE & log_op	'0' & imm12laden & imm12loeschen & regB_imm	addiere & mitCarry & sgnd & mult	'0' & div_starten & sar & shc
101 xxxx	'0' & div_busy	'0' & springeIntr &	ramWElb &	'0' & ovfl & zhm

	& weiter & syncRestart	setzeAdr & springeRel	ramWEhb & csh & cas	& shm
110 xxxx	befehl			
111 xxxx	ip_aus			

Schalter 7 ist mit dem Eingang *Int_Anf_async* verbunden. Steht er auf 1 und ist das IE-Flag gesetzt, wird bei nächster Gelegenheit ein Interrupt ausgelöst.

Für die Eingabe von Daten werden ebenfalls die Schalter S7 bis S0 benutzt, wobei hier S7 auf Bits 15 und 7 geht, S6 auf Bits 14 und 6 etc. High-Byte und Low-Byte des Daten-Eingangs sind somit immer gleich.

Die LEDs über den Tastern werden gemäß Tabelle 2 angesteuert.

Tabelle 2: Leuchtdioden

LED	**Bedeutung**
LED0	ERAMWELowByte
LED1	ERAMWEHighByte
LED2	ERAMLesen
LED3	ExtIOLesen
LED4	ports_we
LED5	frei
LED6	Dparity_fehler
LED7	Bparity_fehler

Die Funktion der Taster geht aus Tabelle 3 hervor.

Tabelle 3: Taster

Taster	**Funktion**
Taster0	1 Takt
Taster1	16 Takte

Taster2	Dauertakten ein/aus
Taster3	Restart

Durch die Synchronisation des Restart-Signals dauert es nach Drücken der Taste 3 zwei Takte, bis *ip_aus* auf 2 steht.

5.3 Testprogramm

Mitgeliefert wird ein Assembler-Programm "test.asm", welches alle Befehle des Prozessors automatisch testet. Im Fall eines Fehlers wird ein Fehlercode an die Ausgabeports 0 und 1 geschrieben, um den Fehler lokalisieren zu können. Dieses Programm wird mittels

```
ssfp16asm test.asm -bp -dp -i
```

assembliert. Die vom Assembler generierten Dateien "test.lst", "test.mem" und "test_edata.mem" müssen (manuell) in das Klassen-Verzeichnis des Simulators kopiert werden (standardmäßig ssfp16sim/class). Mit "java Sim" wird der Simulator gestartet, er lädt automatisch die genannten Dateien.

Um die vom Assembler erstellte Datei "test.mem" in den Speicher des FPGA zu bekommen, ist die Datei "ssfp16.bmm" notwendig. Diese teilt dem im ISE Webpack enthaltenen Programm "data2mem" mit, wie die Daten in "test.mem" zu interpretieren sind und in welche BRAMs sie zu kopieren sind. Das Kopieren geschieht mit

```
data2mem -bm ssfp16.bmm -bd test.mem -bt ssfp16_testumgebung.bit -o b
ssfp16_dat.bit -p xc3s200
```

wobei ssfp16_testumgebung.bit die vom Synthesewerkzeug erstellte Konfigurationsdatei ist. Die fertige Konfigurationsdatei kann mit

```
data2mem -bm ssfp16.bmm -bt ssfp16_dat.bit -d > testconf.txt
```

überprüft werden. Im unteren Bereich befinden sich die Inhalte der BRAMs. Die Datei "ssfp16_dat.bit" kann nun in den FPGA geladen werden. Nach dem Laden befindet sich die Testumgebung im 1-Takt-Modus. Mittels der Tasten S0 und S1 kann also das Programm schrittweise getestet werden.

---

ssfp16

Small simple FPGA 16-bit Microprocessor

Legal

1 introduction and overview

Like many other technicians and engineers I already tinkered in my youth with similar and digital hardware. At that time naturally essentially with individual operation amplifiers, gates and flip-flops. When my father 1990 acquired his first PC, I began with programming. During my study of electro-technology/control engineering to DO Darmstadt became from tinkering increasingly developing, also a micro CONTROLLER and periphery components were used. In order to be able to provide in the context of my first employment real time software for the modelling and regulation of process engineering processes, I needed finally still another portion numeric mathematics, which comes much too briefly from today's viewpoint in the engineer study. Today I concern myself with the integration of safety systems on rail-mounted vehicles, have with the actual hard or software development to thus do no longer much. It is favourable nevertheless, if one understands the technology of the suppliers somewhat more exactly. Programmable logic was one of the few me up to then completely unknown worlds, then I decided to change this. After a few days general orientation (which is a PLA, a CPLD or a FPGA, which software one needs etc.) ordered I the Spartan3-Starter kit of Digilent as

probably one the most favorable Variaten to occur the world of the FPGAs. The first circuits were fast described and so in VHDL should next a digital function table with two channels, 4-zeiliger LCD announcement etc. develop. As far as no problem was noticeable, however that the function table needs actually only a fraction of a FPGAs, which lies for the operation necessary hardware in the FPGA however in the order of magnitude of several 100 Slices plus RAM and which is control comfort nevertheless not that, which one expects from such systems. The planned extension by a USB connection and a A/D transducer was not to be realized on the level anyhow. Only solution was a microprocessor. Of course in the FPGA - that this would be feasible, it was clear me from various articles and forum contributions. Why didn't I merge a finished processor like the PicoBlaze? I wanted nevertheless which to learn - and but am the way as well known the goal.

1.1 considerations to architecture

• The really critical operations with special separate hardware in the FPGA to be settled there can, for very most applications a processor after today's yardsticks of the lower to middle performance class will be sufficient. Who needs a high speed processor, which will fall back to separate processors or DSPs or to FPGAs with inserted processor. Besides I am not reliably able to develop times evenly into a few weeks a High end processor.
• Most recordings of measurement need 10-12 bits, want one still a little to scale, run out it on 16-bit calculations. Audio's applications need likewise 16 bits data capacity per channel.
• The Spartan3-Architektur (SLICEM) suggests to develop a register bank from 16 registers whereby two registers can be read at the same time.
• Instructions should always the same width have, but on the other hand the valuable memory in the FPGA also not waste. 16 bits could reach, in order to code the instruction and up to three operands with in each case 4 bits.
• A modern, small FPGA like the Spartan3-200 has already 12x4 KB memory. The address range should cover thus at least 64 KB.
• The Addressierungsarten directly, directly, indirectly and indirectly with basis is to be supported. These addressing modes are the fundamental and probably by each assembler programmer and by each C-compiler are surely presupposed.
• The points mentioned speak clearly for data bus width of 16 bits. All operations are to refer to 16-Bit of values. Only the letter in RAM should be possible also with 8-bits width.
• Memory gives it in the FPGA quasi arbitrarily scalable, changing code is in the embedded range not usually, thus speaks everything for faster and simpler Harvard architecture.
• It should be possible to be able to address a larger external data RAM or data Rome simply therefore appropriate special instructions should be present.
• In the range of the recording of measurement or regulation frequently multiplications necessary are there, is the processor multiplication instructions to have. A modern FPGA offers fast multipliers - thus optimal conditions.
• Divisions are substantially rarer than multiplications. Only few algorithms need it at all. With the conversion of numbers in decimal notation, how it is frequently needed for the announcement, a repeated subtraction of powers of ten is generally fast enough (on the average about 20-25 clocks per decimal place with speed-optimized programming on the ssfp16). On the other hand a 16-Bit division programmed by means of additions and subtractions needs for arbitrary divisors roughly estimates 100-200 clocks, considers one also if necessary necessary freeing from registers, function calls etc. from therefore instructions for the division of unsigned and signed numbers was planned. Exemplary a relatively small Mehrschritt Dividierer was implemented, which needs 17 clocks. Thus it is only insignificantly faster for the conversion in decimal notation than a repeated subtraction.

• Around periphery efficiently to administer to be able, separate I/O instructions as well as an interrupt input are to be present. • For reliability reasons a parity examination for data and instructions should take place.

1.2 result

With these basic conditions in several iteration steps on the Spartan3 of cut 16-bit processor was developed. Area requirements by only about 200 Slices and a clock frequency could be achieved by very efficient employment of the components of the Spartan3 the available and simultaneous optimization of the coding at the same time by approximately 66 MHz on the slow variant (speed code -4). Here for all instructions with exception of the shop from the memory and the division in principle a clock is needed. Altogether thereby a relationship from speed to surface, substantially better compared with “reproductions” of existing standard processors, results. At the same time it must be naturally pointed out that the basic data mentioned can be achieved probably only with FPGAs of the Spartan3 and Virtex-II families and/or it were not possible to keep all VHDL descriptions hardware independent. At least the WebPack 8.1 is not in some places able to use the hardware from a description of behaviour ideally. If one does not want to code instructions by hand, an assembler must ago. Since I did not find on the fast anything useful (either complicated without any documentation or too simply or too), have I relatively simple assembler in C written. This is described in chapter 3 of this documentation. It is to be compiled hopefully with each existing C-compiler, all the same on which machine. I used both GCC under Linux and Dev C++ with mingw32. Since in a FPGA relatively heavily errors can be searched and I with the VHDL simulation of the processor probably today still to to be simulated would be, I wrote in addition a simulator in Java, which serves at the same time as debugger. The processor was almost illustrated thereby on RTL level, which simplifies the search of errors in the VHDL code, but is naturally for debugging assembler programs unnecessarily slow. Altogether the simulator is really extremely simply held, but but it has also a surface which can be served very simply. In the long run I needed it then nevertheless only for debugging assembler programs, since in the VHDL code from the outset against expecting no substantial errors were. Which is perhaps still missing, is a C-compiler. Since this is not necessary for testing the processor, I did so far to train me also still into adaptable C-compilers. There it surely gives specialists, who get that substantially better and faster than I. Due to the instruction memory limited nevertheless as well as considering the uses of FPGA processors at all a compiler from my view is nevertheless rather “nice ton have” as absolutely necessary.

2 processor

The processor core is described in the file “ssfp16_test.vhd”. The attached” _test " means with the fact only that some signals are additionally led out - on size and speed does not have this influence. The command sentence is tabular represented in ssfp16_befehlssatz.pdf. The ssfp16 is a processor in Harvard architecture, i.e. instructions and data use two completely independent storage areas with independent addressing. The bus structure of the ssfp16-Kerns is represented in fig. 1. The top represents the data path, in the lower part is the command path to be seen. All internal and external control signals including the parity bits are suggested herein only.

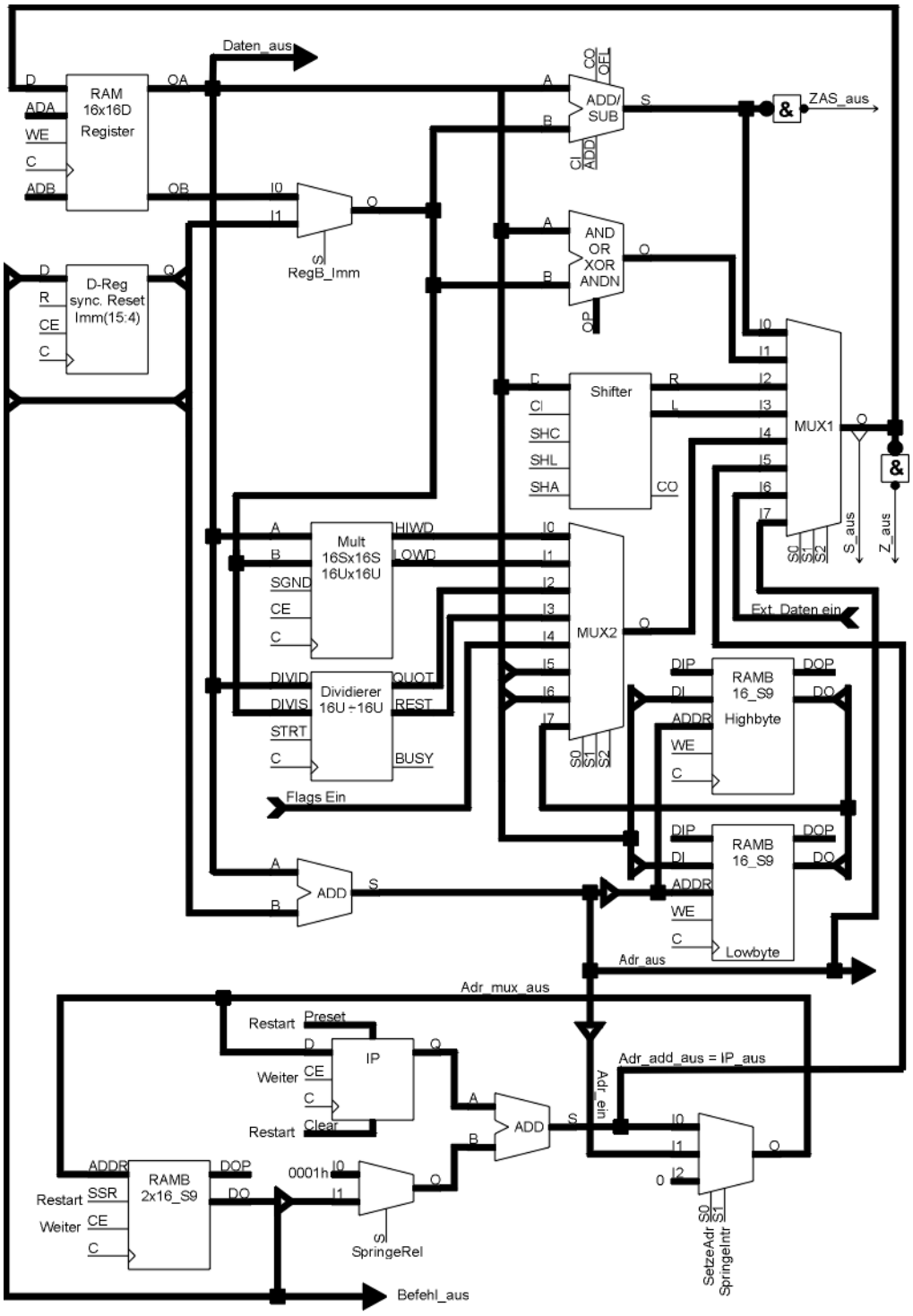


Fig. 1: Bus structure with data and command path.

2.1 overview data path

The data path consists of general registers, the constant register (imm), the units for the treatment of arithmetic and logical operations, the internal data memory, the address adder as well as unites Multiplexern for the selection of the operands and the results. Most of the components are implemented as individual VHDL descriptions, which are hooked up struktural to the data path. Unfortunately the descriptions of the lowest level are partially particularly cut to XILINX FPGAs, there the Synthetisierer (at least ISE Web luggage 8,1) with descriptions of behavior not always optimal results supply.

2.1.1 registers

There are 16 to a large extent equal 16-bit register, r0 to r15 designated. These are called register bank. Some the registers have however a special meaning:
• r0 must always contain the value 0. The assembler forbids therefore each writing access to this register. Processor-moderately this register does not experience any special treatment. • r13 should be used as stack pointer, if a stack is needed. This is however one pure from the user dependent decision, which is given by the assembler neither by the processor nor.
• r14 is defined in the processor as memory place of the return address with interrupts. If no interrupts are used, the register can be used freely.
• r15 is of the assembler as memory place of the return address with call-instructs defined. On the part of the processor just as well registers r8 to r13 for this purpose could be used. The register bank is realized in the form of so-called “distributed RAM” in the FPGA, i.e. SLICEMs are used. Since in many instructions at the same time two registers are needed, the register bank is implemented as dual haven RAM. The write access takes place synchronously with set incoming goods, the read access on both haven asynchronously. To be written can only on haven A. Additionally there are some separate special registers:
• IP - The instruction pointer (Instruction pointer) is realized in form of usual D-flip-flops. It cannot be read directly. If contents are needed exeptionally in the program, a call helps on the next line (without ret). The letter happens by means of jmp or call.
• flags - at present five flags are defined (likewise in the form of D-flip-flops): Bit 0 = transfer (inverted with sub, sbb and cmp), bits 1 = overflow, bit 2 = zero, bit 3 = sign, bit 8 = interrupts permit. Flags can be read and written with movs flags directly.
• multh - the bits 31 to 16 of the result of a multiplication, with preceding imul in complement on two representation. • multl - the bits 15 to 0 of the result of a multiplication, with preceding imul in complement on two representation. multh and multl the output register is physical in the special multiplying unit of the Spartan3. They cannot be set therefore explicitly. • quot - contains the 16-bit quotients of a division. It consists of D-flip-flops.
• remainder - the 16-bit remainder contains of a division. It consists of D-flip-flops. All special registers are 16 bits long.

2.1.2 data flow

With all arithmetic and logical operations including pushing, multiplying and dividing a register of the register bank forms the first operand. This lines up at exit OAU. If a second operand is needed, then this is either a second register of register register (exit WHETHER) or a direct value coded in the instruction. Since in a 16-bit instruction 16-bit value place does not have (finally the instruction must be also still coded), a direct value divided. The lower 4 bits stand thereby always directly in the instruction, in which they are needed. If more than 4 bits are needed, the upper 12 bits in a presented instruction must be coded. These are then stored before a clock in the Imm register. Immediately after the following instruction this register is again put back. The instruction for setting the Imm register is generated exclusively by the assembler and cannot be inserted not manually. After the gate switching time of the individual units the result of the desired operation at the exit O main multiplexers of the MUX1 lines up. With the next rising clock flank it is transferred to the register addressed by ADA, which first operand thus overwritten with the result. A multiplexer is necessary, since the Spartan3-FPGAs does not have Three State logic internally. The moreover one only a 8-zu-1 multiplexer can be developed, needed however at least 14 entrances with one gate level (without bitswap). In order to extend time-critical paths not unnecessarily to describe two 8-zu-1 multiplexers cascaded instead of a 16-zu-1 multiplexer. If a value is to be copied in a RAM or into a peripheral unit, then this in principle from the register addressed by ADA one

takes. The address computation for RAM and periphery is likewise part of the data path. For this the register addressed by ADB is always added to the direct value. In such a way computed (address) value can be written into the register addressed by ADA (see also LEA and MOV further below).

2.1.3 adders/subtracters

The arithmetic operations are:
• ADDs
• sub
• adc
• sbb
• cmp
These operations are accomplished by the adder/subtracter. The Spartan3 supported adders/subtracters in necessary the form ideally in terms of hardware, unfortunately is however available no entsichende collecting main - neither for the Instanziierung nor for the Inferieren. Therefore I described this in accordance with unit the XILINX documents struktural. A functional description would be naturally more beautiful regarding platform independence, but strange to say I did not have an ambition in the place… Note: With subtractions (inclusively cmp) the meaning of the Carry flag is inverted. I.e. if the result is correct, Carry is set. If thus with sbb the Carry flag is not set, additionally 1 is subtracted. This behavior is given of the Spartan3-Hardware so, does not disturb however my judgement also, as long as one does not want to portieren assembler programs with jc-instructions. Then really extreme caution is announced!

2.1.4 logic unit

The logical operations are:
• and
• or
• xor
• andn
• test
The logical operations are processed in a Spartan3-Funktionsblock per bit, which needed exactly the four entrances offers.

2.1.5 sliding unit

Five shift instructions exist:
• srl - logical right pushing (0 after-pushed)
• sra - arithmetic right pushing (MSB after-pushed)
• src - right pushing with Carry (Carry flag after-pushed)
• shl - link pushing (0 after-pushed)
• slc - link pushing with Carry (Carry flag after-pushed)
All instructions shift the exit OAU of the register bank a place. Always both to the right and, the selection of the desired operation is shifted takes place to the left only afterwards by the multiplexer MUX1.

2.1.6 multipliers and Dividierer

In principle an unsigned (mul) and a signed (imul) multiplication are available. According to standard only an unsigned division is implemented, instructions for signed division is however reserved. As operators the same sources are available as for the remaining arithmetic and logical instructions. Both multipliers and Dividierer possess in each case two own 16-bit

output registers, since the results are long 32 bits (multipliers: High Word and Low Word of the result, Dividierer: Quotient and remainder). These registers are changed only by the respective instructions (mul/imul, div/idiv), the results of multiplications and divisions are thus available up to the next multiplication or division. By means of the movs instruction the results can be transferred in the internal registers to the register bank.

2.1.7 BITSWAP, BYTESWAP and SETF

• bitswap - all bits are exchanged (from MSB LSB etc. becomes). This operation is accomplished by the multiplexer MUX2. Admittedly a quite exotic function, which is times needed very rarely however within the Embedded range and by software is very aufwändig. If due to any extension a further MUX entrance were necessary, this instruction would be natural a caper candidate.
• byteswap - High byte and Low byte are exchanged. This operation is accomplished by the MUX2 and is not likewise needed due to the existing byte instructions surely occasionally.
• setf - all flags are set. All three instructions work with exit OAU of the register bank as an operator.

2.1.8 addressing, LEA and MOV

In accordance with specification in the introduction it should be possible to load registers with a constant (direct addressing), to copy values of a register into another register a constant storage location into a register to copy or (direct addressing), one by a register in reverse addressed storage location into a register to copy or in reverse (indirect addressing) as well as one by a register plus a constant addressed storage location in reverse into a register to copy and (indirect addressing with base address). First one could come on the obvious idea to use thus the addition unit to the computation of the address (to introduce no address adder). But then at least the target register would be simultaneous the address register with the bus structure shown in fig. 1, which in many cases would be unpractical. In addition it would be impossible to load directly a constant since this was always added to a register. One would have to always only delete the register thus before (e.g. by an and rx, 0), which would mean an additional instruction in (limited) the ROM and an additional clock. Since it should be also possible to copy a register into another one would have additionally a direct connection of the exit WHETHER to the register bank to the multiplexer to provide. Shop of a constant is however a very frequently needed operation, therefore is a little more logic attached - the separate address adder, which in principle exit WHETHER the register bank to a constant adds. It is 16 bits long. If one leads the result into (arbitrary) a register of the register bank, then one gets lea ry, rx+const instruction. One can add and in any other register store now in addition, register r0 (thus 0) to a constant - which mov rx, const instruction, or however any register adds rx-instruction to 0 and in another register stores - mov ry. Since the arithmetic unit is not used for the addition, flags remain of it unaffected! (One could naturally set using the hardware needed anyway at least sign and zero-flag, that is however not usual.) 2.1.9 storage areas In the FPGA RAM blocks with in each case 4kB are available plus parity bits (BRAM). These are used both for the instruction memory and for the data memory. The word width of the BRAMs is arbitrarily selectable from 1, 2, 4, 8/9, 16/18 and 32/36 bit. If one uses 4 BRAMs with in each case 4 bits in place of the 2 BRAMs thus, then the memory depth without further change can be doubled up to 32 KB (16 BRAMs ever 1 bit long). If one wants to make however a parity examination, in and/or two further BRAMs result. Alternatively one could naturally also always use 9-Bit broad BRAMs and connect in accordance with external multiplexers at the outlet side/the most significant address bits and further-use so the parity bits of the BRAMs even with larger memory. Thus however many different architectures result and in such a way refrained I from it, the memory depth in the case of RAM and ROM generically too described depending upon application purpose and demanded reliability.

Anyway at least the file “ssfp16.bmm” would have to be changed manually. While for the ROM for very simple applications a BRAM (=1024 of instructions) is sufficient, at least two BRAMs must be used, in order to be able to write High byte and Low byte independently for the data memory. Usually processors with more than 8-bits possess data capacity operations for the manipulation of individual bytes. As one can recognize however by the command sentence, the 16 bits of the instructions are to a large extent expenditure-provoked. Even if one wants to thus refer all instructions to bytes, then one would be forced to increase also the command width. Now one could naturally use one of the two parity bits of the ROM still as command bits, but that would be then already a rather unorthodox procedure and also probably with difficulty transferable to other architectures. On the other hand there are only very few points, at which a 16-Bit is value in place of an 8-bit of value of disadvantage. The writing access to RAM one of these few applications might be, would have nevertheless during the writing of a single byte only a word read, which byte which can be overwritten in each case by means of Verundung to 0 are set and set afterwards by Veroderung to the new value. In the worst case that could not have been changed actually byte which can be changed even meanwhile by an interrupt and again with the old value was then overwritten. Therefore the storage areas would have to become very exactly regarded or in principle during such operations interrupts of the program closed. Due to these considerations I decided to plan in principle no byte instructions except to the letter in RAM. Data memory is needed also with many simple applications in relatively large quantities. Here the existing BRAMs frequently not to be sufficient, finally nobody an expensive giant FPGA buy, only in order a few additional KB RAM get. External data memory will be frequently used therefore - it is in form genuine RAM or in form (Flash) of a ROM. In principle three possibilities occur to me of accessing external memory:

1. The external memory is regarded as peripheral equipment, a certain (large) address range is assigned to which. This procedure with micro-control-learn occasionally used, brings however occasionally disadvantages.
2. The external memory is addressed equally to the internal memory with separate instructions and separate address range. This is the standard practice for microprocessors without internal memory. Disadvantage is that additional instructions are necessary.
3. By means of a DMA (direct memory access) CONTROLLER, which writes parallel to the processor into the internal data memory - which would function then at least partly than Cache.

I decided for the second variant, whereby the third variant would be to be integrated due to the dual haven ability of the BRAMs relatively easily additionally. The address computation effected as above described in principle with 16 bits. Thus 64 KB data and/or 65536 instructions can be addressed directly. By means of an address base register in the I/O area nevertheless easily also of any size data memories can be addressed. Altogether thereby the following instructions for reading and writing RAM are available:

- ld - word from internal memory into registers load.
- ldx - word from external memory into registers load.
- st - word of register into internal memory write.
- stb - byte of register into internal memory write.
- stx - word of register into external memory write.
- stxb - byte of register into external memory write.

The ld-instruction needs two clocks, since the BRAMs possesses a register exit, which is only set with the next clock flank. It would be possible to set by disalignment of the clock flanks the BRAM exit in the same processor clock however for this the clock frequency would have to be reduced clearly - which is altogether counter productive. Note: With word instructions the LSB of the address is ignored. Vintages and letters from words thus always takes place with word adjustment. Note: With stb and stxb and LSB of the address = 0 are written the

bits 15 to 8 of the register, with LSB = 1 the bits 7 to 0! Is appropriate for 2,2 command path

2.2.1 command Rome

the size of the ROM available in the FPGA in the form of BRAMs in the order of magnitude of 4-32 KB, typical for micro CONTROLLERs, and should be sufficient for most applications. Otherwise the command path would have to be supplemented around external ROM, which would lead however while maintaining remaining architecture to a clear speed loss. To this case thus at least a instruction Pipelining would have to be introduced, in order to have a whole clock available for the selection of the ROM. The ROM is addressable and only word orientated only by the instruction pointer IP. A Rome address corresponds thus to 16 bits (and/or 18 bits with parities) and an instruction. From the address unit addressably thus 65536 instructions or 128 KB are command Rome. If far one is set, then with the next clock flank the instruction at the lying close address is spent and processed in the same clock (and/or begun with ld and div with processing).

2.2.2 instruction pointers

IP and jumps this register is as broad as for the addressing of the ROM necessarily - with 4 KB = 2048 instructions thus 11 bit. Except with start and restart it contains always the address straight of the instruction standing at the exit of the ROM, not those of the next instruction. If the instruction lining up at the exit of the ROM is no branch instruction and is no interrupt to be implemented, the value of the IP of the address adder is increased by 1 and set on the entrance of the IP and the address entrance of the ROM. If it is a conditioned branch instruction (jxx shortened, whereby xx for condition stands), SpringeRel is set and added thus instead of 1 in the instruction of coded signed 8-bits value. This must be extended naturally before the addition signed to the width of the IP. Thus conditioned jumps to 128 instructions are forward possible back and 127 instructions. Absolute jumps (jmp) and function calls (call) are identical for the command path. In each case by the address adder in the command path by addition of a register and a constant the new address is computed and put at the same time due to laws of the signal setting addr to the entrances IP and ROM. Difference exists only into it that with call, thus if bit 7 is set in the instruction still another incoming goods for the register bank it is set and thus the value of the IP is written plus 1 into the register coded in the instruction - the return address. This register is set by the assembler with a call instruction to 15, with an int instruction to 14, for the processor is possible always always all registers from 8 to 15.

2.2.3 interrupts

for most applications indispensably are interrupts. External interrupt requirements are accepted only if the “interrupt Enable” - flag (IE flag) is set. After start or restart this is not first set, it must only explicitly in the program be set. With call of an external interrupt it is set for 1 to 0, set automatically during the return by means of iret it. The return address is stored effected in register r14 (given by the processor), an automatic safety device from registers on the stack not, since there is no dedicated stack. By the IE flag it is however guaranteed that after an interrupt call only all registers to become secured to be able to make possible without refusing the possibility recursive interrupts. Before an interrupt return interrupts must become if necessary explicitly closed, before the last registers are repaired such as return address and flag - during the return interrupts automatically are then again released. In the command path an interrupt is processed as follows: First far one is taken back by the control and set SpringeIntr. Thus an interrupt clock is inserted - nothing else as a call arranged of the hardware after 0 with storage of the return address in r14. By the cancelling of far one the exit of the address adder on the next instruction which can be required stops - the return address. Inserting the call instruction happens, as a multiplexer in the control gives a call instruction instead of the exit of the ROM (which in this case the instruction already implemented

contains) for further decoding. The target register r14 is likewise generated by the control. Since at address 2 the “correct” program begins (the first address after start and must restart, see below), at address 0 or 1 jmp at the beginning of the interrupt routine. At address 0 an imm instruction will in most cases stand for the upper 12 bits of the branch address. An interrupt clock may not be inserted also with set IE flag at any time: • Not after an imm instruction, since otherwise the imm register was deleted automatically and after the return no more would be available. • Not during an instruction, which needs several clocks (ld, ldx, div), since this was then implemented not completely. • Not if a straight absolute branch instruction (jmp or call) lines up, since due to the cancelling of the far signal the jump was not implemented otherwise at all. Note: Registers MULTL, MULTH, QUOT and REMAINDER cannot be written explicitly. It can therefore in interrupts only then used, if they are not used in the main program or if in the main program for the time, in which the result is needed, the IE flag is in each case set to 0. Since in interrupts actually in principle no complex calculations should be accomplished, this restriction of my judgement does not justify additional, writable registers and appropriate instructions. With the first start (initialization of the FPGA) IP is set for

2.2.4 start and restart

(restart) to 1 and the Rome exit (=Befehl) is set to 0. Same applies to a restart (restart). The Rome exit 0 corresponds to one sub r0, r0, thus an instruction, which does effectively nothing, since r0 is 0 anyway. Because of the address entrance of ROM and IP is then 2, the address of the first instruction. It cannot be excluded surely that the interrupt handling at an instruction address begins smaller 16, then the second instruction must be “more unnecessary” in the assembler program an instruction.

2.3 control

in the module “Steuerung.vhd” all for the controlling of the program and data flow needed signals generated. Also the evaluation of the conditions for conditioned jumps happens in this module (see table 1). The substantial signals for the controlling of the program sequence were already called in the section “command path”. All different might be self-describing. Table 1: Conditioned branch instructions code Mnemonic flag remark 0 jbe emergency C or Z Unsigned 1 jae, jc C Unsigned 2 jb, jnc emergency C Unsigned 3 emergency (emergency C or Z) Unsigned 4 jle (S xor O) or Z Signed 5 jge emergency (S xor O) Signed 6 jl S xor O Signed 7 jg emergency ((S xor O) or Z) Signed 8 js S 9 jns emergency S 10 jo O 11 jno emergency O 14 jz, for each Z 15 jnz, jne emergency Z 2,4 odd-even check the computation and evaluation of the parity bits (parity bits) of the data memory those takes place in the module “datenpfad.vhd”, Evaluation for the instruction memory in the module “befehlspfad.vhd”. Odd parity was selected, since it can be so also easily examined whether a storage location was already initialized. One accesses reading an not initialized storage location, then this contains of 0 in the parity bit and releases thus a parity error. Parity errors are outward given as signal and can release a re-initialisation, an interrupt or a processor change-over there for example. If no odd-even check is needed, this can be removed in the source files specified before and be saved thus approx. 12 LUTs. The Enable entrance of the data memories BRAMs can be then always set to 1. Note: Due to an error in the ISE Web luggage 8,1 is not possible it with this software to initialize the parity bits with only start and restart correctly. Therefore when starting a parity error for command and data memories is always released. Depending upon treatment of the parity errors this must be if necessary considered.

3 assembler

In this section first the assembler language is described, described briefly then the assembler program and finally given a few references to programming.

3.1 assembler language

In principle between “instructions” and “instructions” one differentiates. Instructions are only for the assembler intended, so that this knows, what it to do have. Instructions are defined by the developer of the assembler program and are not given not by the processor. Instructions against it are converted directly from the assembler into machine code. They are given thereby by the processor. Only so-called “pseudo-codes” can be supposed and by the assembler developer to be additionally defined. This are instructions, which are actually special cases of other instructions. Examples are: • mov rx, ry = lea rx, ry+0, • = jmp r15 ret or • = jmp r14 iret Comments begin as usual with “; ” end and at the line end. With instructions and instructions between large and lower case one does not differentiate, however the entire reserved word must be large or small written. A Parser for the computation from expressions to the assembling time is not implemented. If a constant is expected, this cannot thus only be computed from other symbols. A mark must always stand at the start of line. In the address part of instructions the use is optional “+” and/or” - “, it is thus permitted, ld g 1, [r2] in place of ld g 1 to write [r2+0] or stb [27], g 1 in place of stb [r0+27], g 1. 3.1.1 instructions The ssfp16asm knows the instructions represented in table 1. Table 1: Instructions .DATA .EDATA .CODE .ORG December number | Hexzahl | symbol .ALIGN December number | Hexzahl | symbol (1, 2, 4, 8,…, 4096) Mark [:] .EQU December number | Hexzahl [Mark [:]] .DB December number | Hexzahl | symbol | indications | character string [, December number | Hexzahl | symbol | indications | character string]… [Mark [:]] .DW December number | Hexzahl | symbol | indications | character string [, December number | Hexzahl | symbol | indications | character string]… [Mark [:]] .RESB December number | Hexzahl | symbol [Mark [:]] .RESW December number | Hexzahl | symbol .DB and .DW reserve and initialize bytes or Words either in the internal data memory or in the external data memory. Note: Indications and character strings can be initialized also with .DW. In this case a word per indication is written. This is meaningful in connection with odd-even checks (see below) .RESB and .RESW reserve the number of bytes or Words indicated by the following number either in the internal data memory or in the external data memory. With the instructions .DATA, .EDATA and .CODE are communicated to the assembler, to which storage areas the following instructions to refer. Instructions may stand thereby only after .CODE, for memory initializations and - reservations only in .DATA or .EDATA. With .EQU constants are defined. These may be defined within all ranges. .ORG sets the location counter of the current range to the indicated value. This value may not be smaller, than the current value. With .ALIGN the adjustment of the data in the data memory is defined: • .ALIGN 1: Standard adjustment: Byte at byte boundaries, Words at Wordgrenzen. • .ALIGN 2: Also align bytes at Wordgrenzen. • .ALIGN 4, 8,…: All data at DWord/QWord/… - borders align. Worth a power-of-two number must be. At several values behind .DB or .DW the adjustment applies only to the first value, all further is directly attached. Several.ALIGN instructions can stand directly one behind the other. In this case the location counter is set in accordance with the instruction with the largest adjustment, for the following instructions applies the last adjustment value.
Example:

```
.align 0x10
.align 2    var1:
.db 5    text1:
```

.db “that is Text1! ”, 0
text2: .dw “that is Text2! ”, 0
var1 is written one by 0x10 divisible address. text1 and text2 are begun at integral boundaries. Important note: On use of the odd-even check if individual bytes are initialized or written by the program, then the other byte of a word is initialized marked not than. With reading access to the word therefore a parity error is generated. Therefore it must be guaranteed by the programmer that only completely initialized words reading one accesses. This can take place for example by means of renouncement of .DB or with it that behind .DB always a straight number of values stands. In the case of use of the assembler option “- i” must follow accordingly after .RESB a straight number. Without use of the option” - i " must be guaranteed the fact that always only both bytes of a word were written by the program before this word is reading accessed.

3.1.2 instructions

Instructions have the always following syntax: [Mark [:]] Instruction [operand] Instructions with direct operands (e.g. addi, sbbi etc.) are differentiated only by the indicated operands, not by the commandtaken. In place of addi can and must thus ADD moves, imm to be written.

3.1.3 operands

According to their operands can the instructions in groups in accordance with table 2 be divided. These groups of instructions are consulted by the assembler for the syntax check and assignment into the individual action fields. Table 2: Groups of instructions Group of syntax of instructions field assignments OP_RR_RI “instruction reg1, reg2” “instruction reg1, worth”: ADD, sub, adc, sbb, cmp and, or, xor, andn, test mul, imul, div, idiv reg1 => RegA reg2 => RegB_imm4 worth => RegB_imm4 and if necessary Imm12 OP_R “instruction move” shl, slc, sra, srl, src, bitswap, byteswap, setf move => RegA OP_LEA “lea reg1, reg2+wert” lea reg1 => RegA reg2 => RegB worth => Imm4 and if necessary Imm12 OP_MOV “mov reg1, reg2” “mov reg1, worth” mov reg1 => RegA reg2 => RegB worth => RegB_imm4 and if necessary Imm12 OP_MOVS “movs move, regspec” movs move => RegA regspec = multh, multl, remainder, flag quot => Subsubcode OP_LD “instruction reg1, [reg2+wert]” “instruction reg1, [reg2]” “instruction reg1, [worth]” ld, ldx, in reg1 => RegA reg2 => RegB worth => Imm4 and if necessary Imm12 OP_ST “instruction [reg1+wert], reg2” “instruction reg1, [reg2]” “instruction reg1, [worth]” st, stb, stx, stxb, out reg1 => RegB reg2 => RegA worth => Imm4 and if necessary Imm12 OP_CALLJMP “instruction reg+wert” “instruction move” to “worth instruction” call, jmp, int move => RegB worth => Imm4 and if necessary Imm12 call: 15 => RegA int: 14 => RegA jmp: 0 => RegA OP_JXX “instruction worth”, jae, jb, jbe, jg, jge, jl, jle, jz, jnz, ever, jne, jo, jno, js, jns, jc, jnc condition => condition worth => disp NO_OP “instruction” ret, iret 0 => RegA iret: 14 => RegB ret: 15 => RegB 0 => Imm4 If a writing operation is connected with the instruction, then the goal stands always in the first place. If a logical or arithmetic linkage is connected with the instruction, then always applies (a goal: =) Operand1 linkage Operand2. Examples: SUB g 1, R2; G 1: = g 1 - R2 STX [R1+0x1234], R2; external memory [R1+0x1234]: = R2 LD g 1, [R2+0x1234]; G 1: = internal memory [R2+0x1234]

3.2 short description of the assembler program

The assembler reads exactly one entrance file with assembler source text. In accordance with this file are produced

• a file with ending .lst, which contains symbol tables and translated instructions in text form,
• a file with ending .mem, which contains the values, thus a memory map, necessary for the initialization of the FPGA internal data memory and instruction memory,
• and a file with ending _edata.mem, which contains if necessary values for the initialization of external non volatile data memory. In the range .EDATA if no initializations took place, then the file is empty.

3.2.1 options

The following options are available at present:
• - BP: Parity bits for instructions in “name.mem” set.
• - dp: Parity bits for data in “name.mem” set.
• - i: With .RESB and .RESW memory on 0 reserved initializes. If this option is indicated, also the parity bit is set on “initialized”.

3.2.2 structure and function mode

The main program of the assembler is in the file ssfp16asm.c. The lexical characteristics of an assembler source text file are described in the file ssfp16asm.lex. This file is entrance file for the program “flex”, which generates from it a C-source text lex.yy.c. In lex.yy.c is a function yylex (), which goes through the assembler source text file and calls when each recognizing one of the elements an appropriate function described in ssfp16asm.lex. These functions again are defined in lex_fun.c. Already in this stage all initializations of the .DATA and.EDATA ranges are written into the appropriate MEMORY file. After run of yylex () a symbol table and a command table are in the memory. Some the symbols (the marks in the code) must be still updated now, since they were defined only later either when going through the source text file or because by necessary imm instructions marks to have shifted. If all values of the symbols are stable, can be written the instructions with updated values for direct constants into the.mem file. Note: The assembler produces data (.mem file), which are inserted by the program “data2mem” contained in the ISE Web luggage into the FPGA Konfigurationsdatei. These data contain parity bits for internal command and data memory (number of set bits inclusive in the present version in principle. Parity is odd). Due to an error in the version of “data2mem”, supplied with ISE Web luggage 8,1, can be initialized however when using the parity bits of only two thirds of the internal address area. Further values in the.mem file are rated than errors. Therefore must be out-commentated at present in (actually correct) the.mem file manually at least the last third of the start-up values data RAM by means of”/* " and “*”. Thus a third of the command and data memory cannot be initialized naturally in each case.

3.3 references to programming

3.3.1 interrupt routines

It does not take place any automatic safety device from registers. First task of an interrupt is it to store the return address from r14 on the stack to load then the flags into r14 (the only register, which is available in this instant) and from there on the stack to likewise put. Importantly here it is that with the protection of the return address on the stack the flags may not be changed. Thus the pointer of stack may not be degraded yet, therefore [r13-2], r14 st to use and only after copying the flags in r14 the pointer of stack by 4 degrade. With terminating the interrupt accordingly in reverse.

3.3.2 INT instruction

This instruction can be used, in order to call interrupt Handler from the program. The branch destination will be thereby usually not 0. In as far this instruction meaningful is is undecided.

It is to be noted that the IE flag is not deleted automatically by the INT instruction contrary to external interrupts. This must be done therefore manually before call of the interrupt. It is again set by the iret instruction as also with external calls.

3.3.3 NOP instruction

A NOP instruction is not defined, however e.g. lea rx, rx+0 with any register (except r0) has exactly this effect.

4 simulator/debugger

4.1 operation

A condition for the employment is the availability of an Java Runtime environment. One tested with SUN j2re-1.4.2 under Linux and Windows. The simulator is started from the command line with “java Sim”. At present no possibilities for the selection of the files are implemented, therefore the files must be “test.lst”, “test.mem” and “test_edata.mem” in the listing of the simulator classes. Who likes, times a menu border with typical file functions can program. The storage areas for instructions and data in “test.mem” must contain parity information. After the start a surface with three ranges presents itself: Left registers and flags are indicated, in the center contents of the file “test.lst” and within the right range above contents of the internal data memory, under it to that of the external data memory in accordance with contents of the files “test.mem” and “test_edata.mem”. In the center the instruction which can be implemented next is marked at present by means of a blue triangle (the current instruction is not gescrollt automatically however into the visible range). If one clicks with the mouse to the left of a line, a critical point for this line is set, recognizably from the red “stop” - indications. By pressures of “a step” exactly one instruction is implemented, by pressures of “to critical point” the program execution up to reaching a critical point continued. If a line is clicked, it appears grey deposited. With pressures of “to line” the program up to reaching the marked instruction is continued. If the key “RESET” is pressed, the condition is taken after re-initialization of the FPGA, thus as after a only start. The memory files are again read in thereby. Registers can be changed within the left range by input of new values. If the IP register is changed, this expresses itself only after continuation of the program by means of one of the keys “a step”, “to line” or “to critical point”. The value set in the IP register points thereby to the instruction which can be implemented next. Values in the data memory can be changed by input of the address and the new value likewise word orientated. Under Windows the input cursor is set unfortunately not correctly and must be moved therefore by means of mouse on other places. All since the last stop changed registers and storage locations by red writing are marked, all manually changed registers blue.

4.2 structure and function mode

From a detailed description I would like to refrain here, since the simulator/debugger is safe no specimen of trend-setting technology or wants to be - according to carelessly is also the programming style (in some places extremely inefficiently or unergonomisch). The processor was almost illustrated on RTL level, in order to make the search possible of errors in the VHDL code. The files “Processor.java”, “Datenpfad.java”, “Befehlspfad.java” and “Steuerung.java” correspond functionally almost accurately to the VHDL files of the same name. This procedure is naturally relatively complicated, since the temporal parallelism must be considered between the individual modules of the processor exchanged signals. Besides this structure makes a debugging of assembler programs very computer-bound, which becomes quite apparent with longer program runs between stops. For the simulation of

periphery like external RAM, interrupt CONTROLLER etc. is available the module “IOSimulator.java”. At present only external RAM is modelled.

5 installation and tests

5.1 installation

“ssfp16.zip” in any listing unpack.

5.2 test environment

In order to be able to try the processor out really, for example a development board with any Spartan3 (or Virtex II) is necessary. Here I use the board of Digilent, which it for approximately 100 euro zzgl. VAT. in Germany with short delivery times to buy gives. For this board a small test environment is realized, which the announcement of some signals, which makes the gradual processing of the program as well as a releasing possible of interrupts. So the processor can be completely tested. On the 7-Segment-Anzeige with four digits in dependence of the switches S0 to S6 the signals listed in table 1 are represented.

Table 1:7 - segment displays

S6-S0	Digit3	Digit2	Digit1	Digit0
000 0000 - 000 1111	I/O-Ports 0 to 15			
001 xxxx	adr_aus			
010 xxxx	Daten_aus			
011 xxxx	regA_adr	flags(3..0)	'0' & mux_wahl1	'0' & mux_wahl2
100 xxxx	'0' & regWE & log_op	'0' & imm12laden & imm12loeschen & regB_imm	addiere & mitCarry & sgnd & mult	'0' & div_starten & sar & shc
101 xxxx	'0' & div_busy & weiter & syncRestart	'0' & springeIntr & setzeAdr & springeRel	ramWElb & ramWEhb & csh & cas	'0' & ovfl & zhm & shm
110 xxxx	Befehl			
111 xxxx	ip_aus			

Switch 7 is connected with the entrance Int_Anf_async. If it stands on 1 and if the IE flag is set, at next opportunity an interrupt is released. Used for the input of data the switches S7 to S0, whereby S7 goes here on bits 15 and 7, S6 on bits of 14 and 6 etc. High byte and Low byte of the data input are thus always alike likewise. The LEDs over the tracers are headed for in accordance with table 2.

Table 2: Light emitting diodes LED meaning

LED0 ERAMWELowByte

LED1 ERAMWEHighByte

LED2 ERAMLesen

LED3 ExtIOLesen
LED4 ports_we
LED5 free
LED6 Dparity_fehler
LED7 Bparity_fehler
The function of the tracers comes out from table 3.
Table 3: Tracer Tracer function Taster0 1 clock Taster1 of 16 clocks Taster2 lasting documents/out Taster3 restart By the synchronisation of the restart signal it lasts 3 two clocks after pressures of the key, until ip_aus on 2 stands.

5.3 test routine

An assembler program is provided “test.asm”, which tests all instructions of the processor automatically. In the case of an error an error code is written expenditure haven the 0 and 1, in order to be able to locate the error. This program becomes by means of ssfp16asm test.asm - BP - dp - i assembled. The files generated by the assembler “test.lst”, “test.mem” and “test_edata.mem” must be copied (manually) into the Klassen-Verzeichnis of the simulator (according to standard ssfp16sim/class). With “java Sim” the simulator is started, it loads automatically the files mentioned. In order the file “test.mem” into the memory of the FPGA, provided by the assembler, too gotten, is necessary the file “ssfp16.bmm”. This communicates to the program “data2mem” contained in the ISE Web luggage, how the data in “test.mem” are to be interpreted and into which BRAMs them be copied are. Copying happens also
data2mem - CBM ssfp16.bmm - bd test.mem - ssfp16_testumgebung.bit bt - o b ssfp16_dat.bit - p xc3s200
whereby ssfp16_testumgebung.bit is the configuration file provided by the synthesis tool. The finished configuration file can also
data2mem - CBM ssfp16.bmm - bt ssfp16_dat.bit - D > testconf.txt
are examined. In the lower range are contents of the BRAMs. The file “ssfp16_dat.bit” can be loaded now into the FPGA. To the shop the test environment is in the 1-Takt-Modus. By means of the keys S0 and S1 can be tested thus the program gradually.