TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät für Elektrotechnik und Informationstechnik Institut für Grundlagen der Elektrotechnik und Elektronik

Professur für hochparallele VLSI-Systeme und Neuromikroelektronik

BELEGARBEIT VLSI-PROZESSORENTWURF

Adrian Zinke

Studienrichtung: ET
Matrikelnummer: 3905879

Timo Nicolai

Studienrichtung: IST
Matrikelnummer: 4048209

SVN-Gruppennummer: Gruppe02

15. August 2018

INHALTSVERZEICHNIS

Selbstständigkeitserklärung

Hiermit versichern wir, dass wir die vorliegende Arbeit mit dem Titel *Belegarbeit VLSI-Prozessorentwurf* selbstständig und ohne unzulässige Hilfe Dritter verfasst haben. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Uns ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 15. August 2018

Adrian Zinke

Timo Nicolai

1 EINLEITUNG

Mikroprozessoren sind aus unserer heutigen Gesellschaft nicht mehr wegzudenken. Man findet sie in Handys, PCs, Autos und Herzschrittmachern, um nur einige wenige Beispiele zu nennen. Im Modul *VLSI-Prozessorentwurf* bestand die Aufgabe darin, einen Teil des Entwicklungszyklus eines Mikroprozessors anhand des Beispiels einer mit Zilog Z80 Bytecode kompatiblen integrierten Schaltung nachzuvollziehen. Dies beinhaltet deren Entwurf, Implementierung sowie Verifikation per Simulation und anschließende Synthese und Layout.

In diesem Bericht wird zuerst auf die von uns verfolgten Design-Ziele und anschließend auf die Struktur unseres Verilog-Codes und die Implementierungsdetails der wichtigsten Module eingegangen. Anschließend folgt eine Betrachtung des von unserer Implementierung während Speicher- und IO Zugriffen gezeigten Timings. Danach wird auf die Verifikation und dabei insbesondere auf unsere Ansätze zur Automatisierung des Verifikationsprozesses eingegangen. Das darauffolgende Kapitel beschreibt die von uns getroffenen Maßnahmen um unsere Implementierung bezüglich der von uns gesetzten Design Ziele zu optimieren und den dabei erzielten Erfolg. Anschließend werden die Ergebnisse der Synthese und Place & Route Schritte präsentiert. Zuletzt erfolgt eine kurze Einschätzung und Bewertung der Arbeit.

2 DESIGN-ZIELE

Oberste Priorität hat die korrekte Funktionalität unserer Implementierung. Entsprechend ist ein großer Teil des Entwicklungsaufwandes in die Verifizierung geflossen.

Als Optimierungsziel haben wir uns auf die Geschwindigkeit des Prozessors festgelegt. Die erreichte Geschwindigkeit hängt dabei sowohl von der maximalen Taktfrequenz der resultierenden Schaltung als auch der Länge der einzelnen Instruktionen in Takten ab. Das exakte Optimierungsziel ist hier ohne Betrachtung konkreter Benchmark-Programme nicht klar definiert, da das Einfügen zusätzlicher Takte in einige Instruktionen unter Umständen eine Erhöhung der Taktfrequenz ermöglicht, wenn hierbei zusätzliche Register auf dem kritischen Pfad eingefügt werden können. Hier lässt sich also maximal ein Pareto-Optimum erreichen. Das Anlegen von Benchmark-Programmen ist wiederum nicht trivial, da nicht bekannt ist, in welchen Bereichen der entwickelte Prozessor eingesetzt werden soll. Beispielsweise ist es schwierig zu entscheiden, ob die Geschwindigkeit der 16-Bit ALU-Operationen genauso wichtig ist wie die der 8-Bit ALU-Operationen. Dies kann aber zum Beispiel die Entscheidung für oder gegen eine 16-Bit ALU beeinflussen.

Wir haben mit dem Ansatz gearbeitet, zunächst alle Instruktionen so zu implementieren, dass ihre Länge in Takten dem unteren Limit entspricht, das durch die während der Instruktion notwendigen Speicherzugriffe bestimmt ist. Anschließend wurde unter Beibehaltung dieser Instruktions-Längen die Taktfrequenz optimiert. Erst danach wurde versucht, durch Einfügen zusätzlicher Takte die Taktfrequenz noch weiter zu erhöhen. Ob der dabei erzielte Taktfrequenz-Gewinn die Verlängerung der betroffenen Instruktionen wert ist wurde dann von Fall zu Fall nicht formal sondern rein intuitiv entschieden. Die erzielten Ergebnisse sind in Abschnitt 6 dokumentiert.

3 MODULBESCHREIBUNGEN

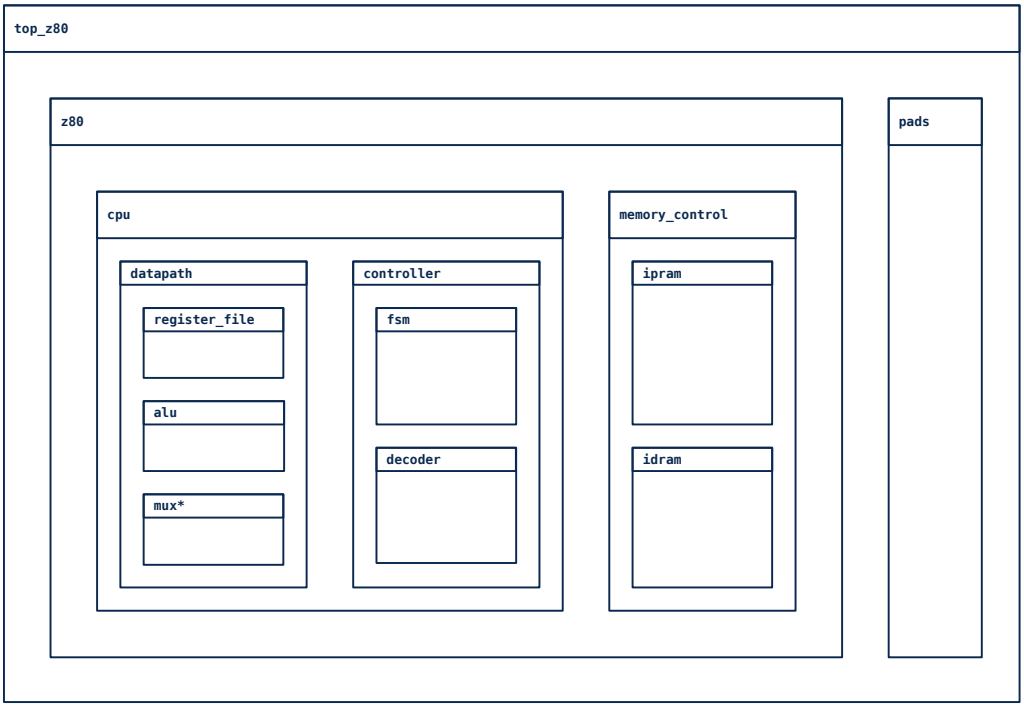

Abbildung 3.1: Modulstruktur

Die Modulstruktur des Verilog Codes unserer Z80-Implementierung ist in Abbildung 3.1 gezeigt. Das Toplevel ist das Modul `top_z80`, in diesem sind der eigentliche Prozessor (Modul `z80`) und die Pads instanziiert.

Das Modul `z80` enthält die CPU (Modul `cpu`) und die Speicherverwaltung (Modul `memory_control`). Alle Speicherzugriffe der CPU werden an die Speicherverwaltung weitergeleitet. Diese spricht wiederum das korrekte Speicherelement an. Der interne Programm- und der interne Datenspeicher sind in den Modulen `ipram` und `idram` implementiert, welche dünne Wrapper um die verwendeten proprietären Speicher-Zellen sind. Beide sind im `memory_control` Modul instanziiert.

Die CPU besteht aus dem Datenpfad (Modul `datapath`) und einem zugehörigen Controller (Modul `controller`), welcher die Steuersignale für den Datenpfad generiert.

Die wichtigsten Elemente im Datenpfad sind das Register File (Modul `register_file`) und die ALU (Modul `alu`). Das Register File kapselt die 8- und 16-Bit CPU-Register, die ALU implementiert alle arithmetisch-logischen Operationen und die Generierung der Flags.

Der Controller besteht aus der FSM (Modul `fsm`) und dem Decoder (Modul `decoder`). Die FSM ist der Zustandsautomat des Prozessors, der den Ablauf der Befehlsabarbeitung koordiniert. Der Decoder generiert unter Kenntnis des aktuellen Zustandes und der auszuführenden Instruktion kombinatorisch die benötigten Steuersignale für den Datenpfad.

Fast alle Module binden per `` `include `` Statements "Header-Files" ein, in denen von mehreren

Modulen geteilte lokale Parameter definiert sind. Lediglich die Busbreiten von Modul Ein- und Ausgängen sind in der Datei `buswidth.v` der Einfachheit halber über `‘define` Statements global definiert.

Die folgenden Modulbeschreibungen beziehen sich bewusst auf den Stand der Implementierung vor dem finalen Optimierungsschritt, welcher hierdurch besser nachvollziehbar wird. Das heißt, dass die Beschreibungen an einigen wenigen Stellen leicht vom eingereichten Verilog-Code abweichen können. Dies sollte die Verständlichkeit der Ausführungen jedoch nicht beeinträchtigen. Die im finalen Optimierungsschritt vorgenommenen Änderungen werden dann im Abschnitt 6 erläutert.

3.1 CPU

Das `cpu` Modul ist das “Herz” der Implementierung. Prinzipiell ist dieses Modul die eigentliche Implementierung der Dokumentation, Memory Control und Pads stellen lediglich die Schnittstellen zu On-Chip Speicher und zur “Außenwelt” dar.

3.1.1 DATENPFAD

Abbildung 3.2 zeigt den schematischen Aufbau des Datenpfades. ALU and Register-File werden in eigenen Abschnitten im Detail erläutert.

Im Datenpfad sind neben Register-File und ALU eine Reihe alleinstehender Register, sowie mehrere Multiplexer instanziiert. Die alleinstehenden Register sind alle Instanzen des `ff` Moduls, welches einen einfachen D-Flipflop beschreibt.

Zu jedem Multiplexer existiert hingegen ein eigenes Modul. Einige der in Abbildung 3.2 als Multiplexer-Eingänge dargestellten Signale werden dabei teilweise erst in diesen Multiplexer-Modulen aus anderen Signalen zusammengesetzt. Dies ist in Listing 3.1 am Beispiel des Mutiplexers für den ALU-Operanden B gezeigt. Unter anderem können der konkatenierte Inhalt der beiden Dateneingangs-Puffer `reg_mem_din_hi` und `reg_mem_din_lo`, der vorzeichenerweiterte Wert des `reg_mem_din_lo` Puffers oder der konstante Wert zwei (oft zur Inkrementierung des PC vor dessen Ablage auf dem Stack benötigt) ausgegeben werden.

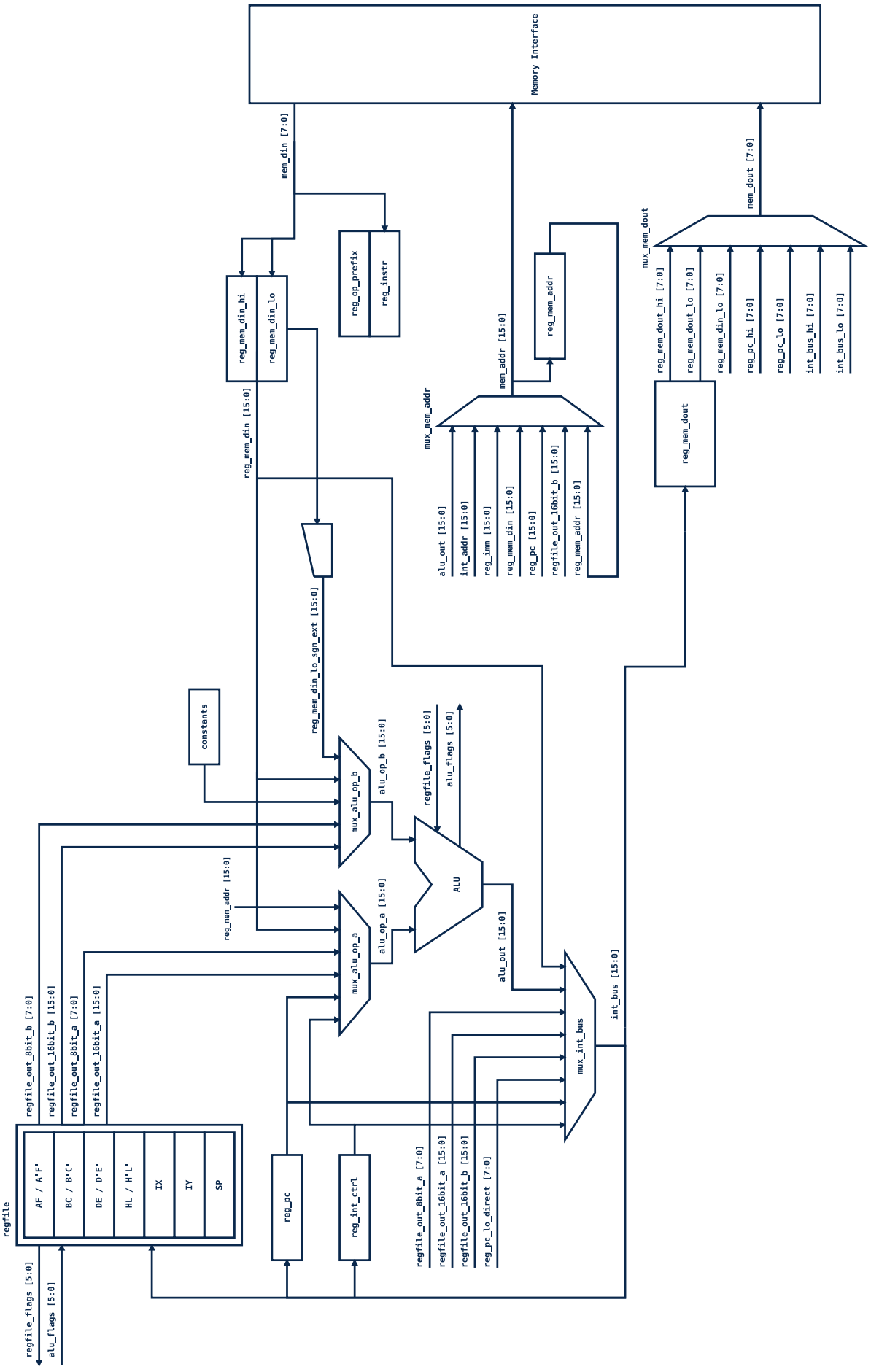

Abbildung 3.2: Datenpfad

Listing 3.1: Alu-Operand B Multiplexer (mux_alu_op_b.v)

```
`timescale 1ns/10ps

`include "buswidth.vh"

module mux_alu_op_b (
    input [`MUX_ALU_OP_B_SEL_WIDTH-1:0] data_select,
    input [7:0] regfile_out_8bit_b,
    input [15:0] regfile_out_16bit_b,
    input [7:0] reg_mem_din_hi,
    input [7:0] reg_mem_din_lo,
    output reg [15:0] data_out
);

localparam MUX_ALU_OP_B_IDX_REGFILE_OUT_16BIT_B    = 0;
localparam MUX_ALU_OP_B_IDX_REG_MEM_DIN            = 1;
localparam MUX_ALU_OP_B_IDX_REG_MEM_DIN_LO_SGN_EXT = 2;
localparam MUX_ALU_OP_B_IDX_CONST2                 = 3;

always @* begin
    data_out = { 8'h00, regfile_out_8bit_b };

    if (data_select[MUX_ALU_OP_B_IDX_REGFILE_OUT_16BIT_B])
        data_out = regfile_out_16bit_b;
    if (data_select[MUX_ALU_OP_B_IDX_REG_MEM_DIN])
        data_out = { reg_mem_din_hi, reg_mem_din_lo };
    if (data_select[MUX_ALU_OP_B_IDX_REG_MEM_DIN_LO_SGN_EXT])
        data_out = { {8{reg_mem_din_lo[7]}}, reg_mem_din_lo };
    if (data_select[MUX_ALU_OP_B_IDX_CONST2])
        data_out = 16'h0002;
end

endmodule
```

Der data_select Eingang ist 1-aus-n kodiert, daher kann das Ausgangs-Signal mit der gezeigten Kaskade von if-Statements selektiert werden. Dies ist gerade bei Multiplexern, die zum kritischen Pfad gehören, wichtig, da die Propagierung von Signalen durch den synthetisierten Multiplexer so leicht beschleunigt wird (siehe auch Abschnitt 6). Da der entstehende Flächen-Overhead vernachlässigbar ist, wurde diese Optimierung der Einfachheit halber in allen Multiplexer Modulen vorgenommen.

Wesentliches Merkmal des Datenpfades ist der interne 16-Bit Bus int_bus. Über den mux_int_bus Multiplexer können unter anderem alle Register aus dem Register-File und die Ausgabe der ALU auf diesem Bus ausgegeben und folgend in eines der Register des Register-Files, den Program Counter, das Interrupt Control Register oder den 16-Bit Datenausgabe-Puffer reg_mem_dout übernommen werden. Getrennte Multiplexer sind hier nicht notwendig, da diese Register nie im gleichen Takt beschrieben werden. 8-Bit Werte (zum Beispiel das Ergebnis einer 8-Bit ALU-Operation) werden im unteren Byte des internen Datenbusses transportiert.

Der Multiplexer-Eingang reg_pc_lo_direct wird vom Decoder während nicht-maskierbarer Interrupts, maskierbarer Interrupts bei gesetztem Interrupt-Modus 1 und RST Instruktionen auf das untere Byte der (konstanten) Adresse gesetzt, zu der folgend gesprungen werden muss (das obere Byte dieser Sprungadressen ist in jedem Fall null).

Zu beachten ist, dass es keine separaten arithmetischen Elemente zur Inkrementierung des Program Counters oder zur Inkrementierung bzw. Dekrementierung des Stack Pointers gibt, diese Aufgaben übernimmt die ALU[1].

Aus dem Register-File können parallel zwei beliebige 8- sowie 16-Bit Register ausgelesen werden. Es werden in keinem Fall im gleichen Takt sowohl ein 8- als auch ein 16-Bit Register benötigt, die Trennung der Register-File Ausgänge "a" und "b" in 8- und 16-Bit Varianten ist lediglich eine Performance-Optimierung, welche die kombinatorische Verzögerung durch das

[1] Das 16-Bit Register BC kann jedoch auch ohne die ALU dekrementiert werden, mehr dazu in Abschnitt 3.1.1.2.

synthetisierte Register-File bei der Register-Auswahl reduziert.

Die ALU führt 8- und 16-Bit Datenoperationen aus, unter anderem Addition, Subtraktion, Setzen und Löschen von Bits sowie Schiebeoperationen. Die Operanden einer ALU-Operation werden über die Multiplexer `mux_alu_op_a` und `mux_alu_op_a` an die ALU angelegt. Einige ALU-Operationen verwenden dabei nur einen dieser Operanden.

Die ALU arbeitet prinzipiell immer mit 16-Bit Operanden und gibt ein entsprechendes 16-Bit Ergebnis aus. Für 8-Bit Operationen werden nur die unteren Bytes beider Operanden genutzt und nur das untere Byte des Ergebnisses verwertet.

Der Inhalt des Flag-Registers liegt permanent über den `regfile_flags` Bus an der ALU an. Während jeder ALU-Operation werden die aktualisierten Flags über den `alu_flags` Bus ausgegeben, der wiederum Eingang des Register-Files ist. Ist ein entsprechendes Kontrollsignal gesetzt, wird dieser Wert mit steigender Taktflanke ins Register-File übernommen.

Die Kombination von internem 16-Bit Bus und 16-Bit ALU (im Gegensatz zur Nutzung einer entsprechenden 8-Bit ALU) erlaubt es unter anderem, 16-Bit ALU-Operationen und Adressberechnungen in nur einem Takt auszuführen, auf Kosten einer größeren kombinatorischen Verzögerung durch die ALU in der synthetisierten Schaltung.

3.1.1.1 SPEICHER-INTERFACE

Die drei Busse, die die Schnittstelle zwischen Datenpfad und Speicher bzw. externen Geräten darstellen, sind in Abbildung 3.2 mit dem mit *Memory-Interface* beschrifteten Block (der keinem Modul entspricht) verbunden.

Beim lesenden Speicherzugriff wird die Adresse, von der gelesen werden soll, über den `mux_mem_addr` Multiplexer auf den `mem_addr` Bus ausgegeben. Da diese hierbei stets über mehrere Takte gehalten werden muss (siehe auch Abschnitt 4), ist hinter den Multiplexer das Register `reg_mem_addr` geschaltet, das wiederum ein Eingang des Multiplexers ist. In einem typischen Anwendungsfall kann so in einem Takt eine von der ALU berechnete Adresse auf `mem_addr` ausgegeben und gleichzeitig mit der nächsten steigenden Taktflanke in `reg_mem_addr` übernommen werden. In folgenden Takten kann dann der Wert in `reg_mem_addr` zur Speicher-Adressierung genutzt werden, während die ALU andere Operationen ausführt.

Das Signal `int_addr`[2] wird im Multiplexer generiert und bei Auftreten von Interrupts im Interrupt-Modus 2 genutzt. Das Signal setzt sich aus dem Wert des `reg_int_ctrl` Registers und dem extern eigelesenen, in `reg_mem_din_lo` gepufferten unteren Byte der Interrupt-Handler-Adresse zusammen (das niederwertigste Bit wird dabei immer genullt um 16-Bit Alignment dieser Adresse zu garantieren).

Das Signal `reg_imm` wird ebenfalls im Multiplexer generiert und setzt sich aus dem Wert des Akkumulators (über `regfile_out_8bit_b`) und einem aus dem Speicher gelesenen Immediate Byte zusammen. Es wird während den Instruktionen `IN A,(n)` und `OUT (n),A` zur Adressierung von Geräten benötigt.

Wird während eines Instruction Fetches ein Opcode aus dem Speicher gelesen, wird dieser im `reg_instr` Register gespeichert. Viele Instruktionen haben zusätzlich einen -maximal zwei Byte langen- *Präfix*, der die Bedeutung des Opcodes verändert. Der Decoder erkennt während eines Instruction Fetch automatisch, dass das geladene Byte einen solchen Präfix darstellt und dieser wird im Register `reg_op_prefix` festgehalten. Dieses kodiert dabei auf kompakte Art sowohl ein- als auch zwei-Byte Präfixe. Ein eingelesenes Präfix-Byte kann demnach nicht einfach in das Register übernommen werden. Stattdessen muss vom Decoder ein entsprechender neuer Wert anhand des eingelesenen Präfix-Bytes und eines möglicherweise bereits in `reg_op_prefix` kodierten, zuvor eingelesenen, Präfix-Bytes bestimmt werden. Die Inhalte

[2] Achtung bei der Namensgebung: viele im Zusammenhang mit Interrupt-Behandlung stehenden Signale haben den Präfix "`int`" mit dem internen Datenbus `int_bus` als einiziger Ausnahme.

beider Register werden an den Controller weitergereicht, der sie nutzt um Entscheidungen bezüglich Zustandsübergängen und der Erzeugung von Kontrollsignalen zu treffen.

Eingelesene Datenbytes werden hingegen in einem der Register `reg_mem_din_hi` und `reg_mem_din_lo` gepuffert. Beim Lesen von 16-Bit Werten aus dem Speicher in zwei aufeinanderfolgenden Leseoperationen kann das höherwertige Byte in `reg_mem_din_hi` und das niederwertige Byte `reg_mem_din_lo` gepuffert werden. Der durch Konkatenation der Register-Inhalte enstehende 16-Bit Wert kann dann direkt auf den internen Bus ausgegeben oder als Operand an die ALU angelegt werden.

Zusätzlich kann auch der auf 16-Bit vorzeichenerweiterte Wert des `reg_mem_din_lo` Puffers (hier mit `reg_mem_din_lo_sgn_ext` bezeichnet) als Operand B an die ALU angelegt werden. Während Instruktionen, die mit indizierter Adressierung arbeiten, wird der entsprechende Offset in diesen Puffer geladen und direkt im nächsten Takt vorzeichenerweitert zum Inhalt eines der Register `IX` oder `IY` addiert. Das Ergebnis kann sofort zur Speicheradressierung genutzt werden. Hierdurch wird ein zusätzlicher Maschinen-Zyklus eingespart, in dem sonst lediglich eine Adressberechnung und kein Speicherzugriff durchgeführt werden müsste.

Für schreibende Speicherzugriffe wird das zu schreibende Byte auf dem Bus `mem_dout` ausgegeben. Hier wird ebenfalls ein Puffer eingesetzt, jedoch vor dem Datenausgangs-Multiplexer `mux_mem_dout`. In diesem können über den internen Bus 16-Bit Werte (z.B. von der ALU berechnete) gepeichert und dann in zwei aufeinanderfolgenden Takten in den Speicher geschrieben werden.

3.1.1.2 REGISTER FILE

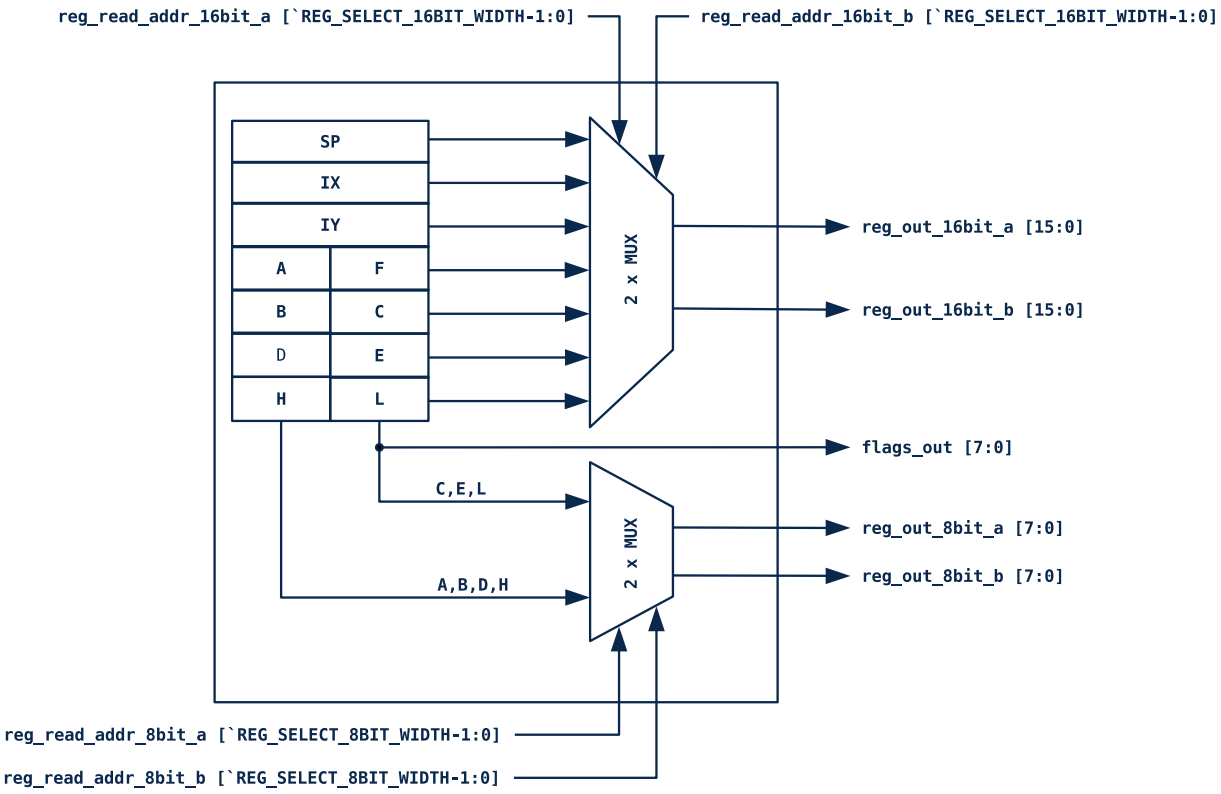

Abbildung 3.3: Lesender Register-File Zugriff

Das Register File besteht aus drei 16-Bit Registern, acht 8-Bit Registern sowie acht 8-Bit Schatten-Registern.

Abbildung 3.3 zeigt schematisch die für einen lesenden Zugriff relevanten Komponenten des Register-Files. Das Flag-Register ist direkt mit dem Ausgang `flags_out` verbunden und somit

permanent verfügbar. Dabei werden die zwei undefinierten Flag-Bits des 8-Bit breiten Flag-Registers nicht ausgegeben. Diese werden in unserer Implementierung auch nicht zum Speichern temporärer Information o.ä. verwendet und können nicht direkt beschrieben werden. Über die vier `reg_read_addr*` Eingänge kann gesteuert werden, welche Register an den vier entsprechenden Ausgängen ausgegeben werden. Die 8-Bit Register `A` und `F`, `B` und `C`, `D` und `E`, `H` und `L` sind einzeln als 8-Bit Register adressierbar, lassen sich aber auch paarweise über einen der 16-Bit Ausgänge ausgeben. Die Schattenregister lassen sich nicht auslesen und sind in Abbildung 3.3 nicht gezeigt.

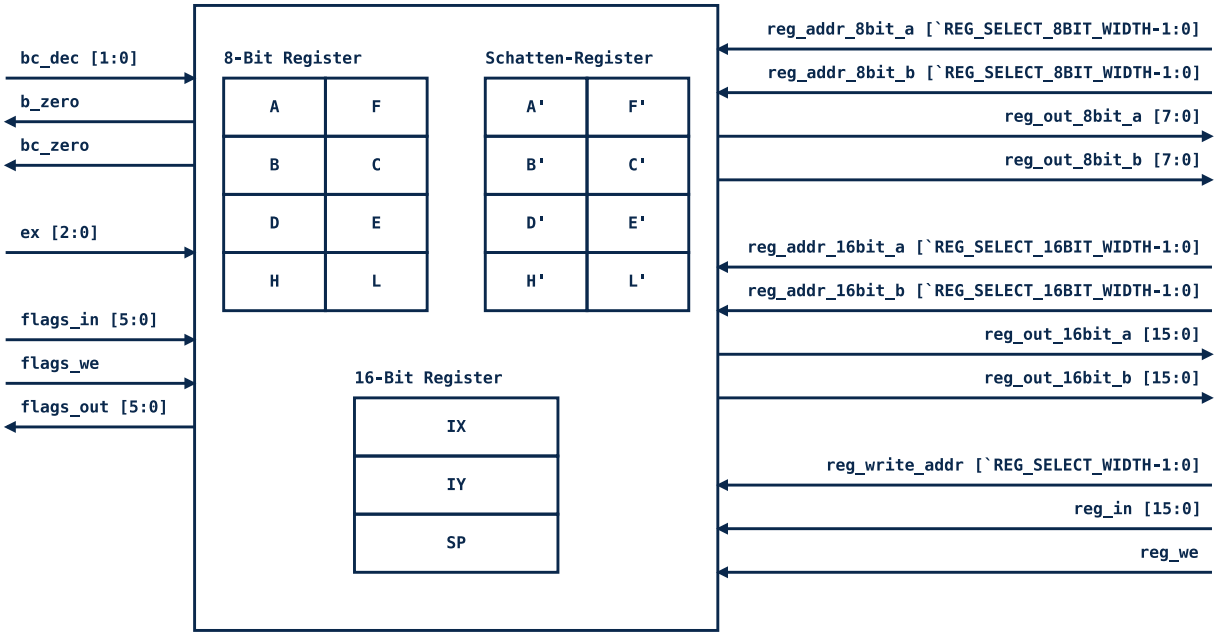

Abbildung 3.4: Register-File

Abbildung 3.4 zeigt schematisch alle Register-File Ein- und Ausgänge. Um ein Register zu beschreiben müssen die Daten an den 16-Bit Eingang `reg_in` angelegt werden. Das Zielregister für den Schreibzugriff wird mit Hilfe des Signals `reg_write_addr` ausgewählt. Im Gegensatz zu den `reg_read_addr*` Eingängen können über diesen Eingang sowohl 8- als auch die 16-Bit Register angesprochen werden. Ist der Write-Enable-Eingang `reg_we` aktiv, wird mit der nächsten steigenden Taktflanke der anliegende Wert übernommen. Für 8-Bit Register ist dies stets das untere Byte des an `reg_in` anliegenden 16-Bit Wortes. Für das Flags-Register stehen der separate Eingang `flags_in` und Write-Enable-Eingang `flags_we` zur Verfügung.

Weiterhin verfügt das Registerfile über einige besondere Funktionalitäten:

- Der `regfile_bc_dec` Eingang wird während der Abarbeitung der Block-Transfer- und Such-Instruktionen genutzt um ohne Nutzung der ALU den Byte-Counter, d.h. das 16-Bit Register `BC` zu dekrementieren. Dadurch müssen keine zusätzlichen Zustände ohne Speicherzugriff eingeführt oder ALU-Operationen außerhalb des letzten Taktes eines Maschinen-Zyklus ausgeführt werden (siehe auch Abschnitt 3.1.2.2).

  Es wird hierfür zwischen zwei Modi unterschieden, `BC_DEC_LD` und `BC_DEC_CP`. Der Eingang wird während der Instruktionen `LDI`, `LDIR`, `LDD` und `LDDR` auf ersteren gesetzt, während der entsprechenden `CP*` Instruktionen auf letzteren. In beiden Fällen wird das `BC` Register dekrementiert. Die Unterscheidung ist notwendig, da gleichzeitig auch passende Änderungen am Flag-Register vorgenommen werden. Für alle anderen Instruktionen wird der Eingang auf `BC_DEC_NONE` gesetzt.

- Der `regfile_b_zero` Ausgang des Register-Files ist immer dann high wenn das 8-Bit Register `B` den Wert null hat. Er wird während der `DJNZ` Instruktion genutzt um die Sprungentscheidung zu treffen.
- Der `regfile_bc_zero` Ausgang des Register-Files ist entsprechend dann high wenn das 16-Bit Register `BC` den Wert null hat und wird während der Instruktionen `LDIR`, `LDDR`, `CPIR` und `CPDR` gebraucht, die in diesem Fall abgebrochen werden können.
- Mit dem 1-aus-n kodierten `ex` Eingang kann in einem Takt der Inhalt von Akkumulator und Flag-Register (`EX_AF`) oder der **aller** 8-Bit Register (`EX_EXX`) mit dem der entsprechenden Schatten-Register getauscht werden. Alternativ können auch 16-Bit Register `DE` und `HL` vertauscht werden (`EX_DE_HL`). Damit können die verschiedenen Exchange Intruktionen realisiert werden.

3.1.1.3 ALU

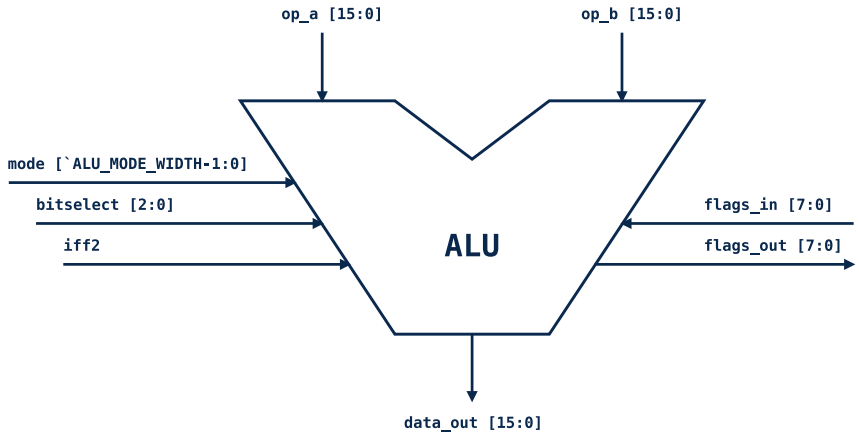

Abbildung 3.5: ALU

Abbildung 3.5 zeigt schematisch alle Eingänge- und Ausgänge der ALU. Über den `mode` Eingang kann der Controller die von der ALU auszuführende Operation festlegen, dies beinhaltet auch die Generierung der entsprechenden Flags, die wie zuvor beschrieben über `alu_flags` ausgegeben werden (nicht modifizierte Flags werden einfach von `regfile_flags` zu `alu_flags` durchgereicht). Einige wenige Instruktionen modifizieren die Flags ohne eine damit zusammenhängende ALU-Operation auszuführen, wie etwa die `IN` Instruktion, welche die Flags allein in Abhängigkeit eines von einem externen Gerät eingelesenen Bytes setzt. Für diese Fälle werden dennoch separate ALU-Modi definiert. Für die genannte Instruktion muss hier z.B. das eingelesene Byte als Operand A an die ALU angelegt werden und die ALU ist im gleichen Takt nicht für eventuell parallel notwendige Berechnungen verfügbar. In unserem Design entstehen hierduch jedoch an keiner Stelle Timing-Probleme.

Über den `bitselect` Eingang kann bei Bit-Set, -Reset und -Test Operationen spezifiziert werden auf welchem der 8-Bits des Operanden A die Operation ausgeführt werden soll.

Über den `iff2` Eingang liegt permanent der aktuelle Wert des `iff2` Registers (im Modul `fsm` definiert) an der ALU an. Dieser wird nur während der `LDAI` Instruktion benötigt, die außer der Inkrementierung des PC keine ALU-Operation ausführt, aber den Wert von `iff2` über `alu_flags` als aktualisiertes Partitäts-Flag ausgibt.

Alle arithmetischen ALU-Operationen werden intern über eine 16-Bit Addierer-Beschreibung realisiert, die in Listing 3.2 gezeigt ist. Für Subtraktionen nimmt `b` hierbei den invertierten Wert von `op_b` an. Der eingehende Carry `cin0` wird in Abhängigkeit davon gesetzt, ob die auszuführende Operation eine Subtraktion beinhaltet und ob diese das gesetzte Carry-Flag

berücksichtigt. Der Addierer setzt sich aus mehreren Addierer-Beschreibungen variabler Bitbreite und zwei Full-Adder Beschreibungen zusammen. Dies ist notwendig um eine Reihe von eingehenden und ausgehenden Carries zu bestimmen. Anhand dieser werden für viele arithmetische Operationen die resultierenden Carry- und Overflow-Flags bestimmt. Der ausgehende Carry des dritten Bits, `cout3`, ist der Half-Carry bei 8-Bit Additionen, bzw. das Inverse des Half-Borrows bei 8-Bit Subtraktionen. `cout11` ist das Pendant für einige 16-Bit Operationen. Mit `cin7` und `cout7` können Carry und Overflow für 8-Bit Operationen, mit `cin15` und `cout15` für 16-Bit Operationen bestimmt werden.

Listing 3.2: 16-Bit Adder Implementierung (Ausschnitt `alu.v`, angepasste Formattierung)

```
// ...

sub = ((mode == ALU_MODE_SUB)       || (mode == ALU_MODE_SBC) ||
       (mode == ALU_MODE_SBC_16BIT) || (mode == ALU_MODE_CP)  ||
       (mode == ALU_MODE_NEG)       || (mode == ALU_MODE_CPB));

carry = ((mode == ALU_MODE_ADC) || (mode == ALU_MODE_ADC_16BIT) ||
         (mode == ALU_MODE_SBC) || (mode == ALU_MODE_SBC_16BIT));

cin  = flags_in[FLAG_IDX_C];
cin0 = sub ^ (carry && cin);

// ...

// perform calculations
case (mode)

    // arithmetic operations
    default: begin
        { cout3, data_out[3:0] } = { 1'b0, a[3:0] } + { 1'b0, b[3:0] } + { 4'b0000, cin0 };

        { cin7, data_out[6:4] } = { 1'b0, a[6:4] } + { 1'b0, b[6:4] } + { 3'b000, cout3 };

        data_out[7] = (a[7] ^ b[7]) ^ cin7;
        cout7 = (cin7 & (a[7] ^ b[7])) | (a[7] & b[7]);

        { cout11, data_out[11:8] } = { 1'b0, a[11:8] } + { 1'b0, b[11:8] } + { 4'b0000, cout7 };

        { cin15, data_out[14:12] } = { 1'b0, a[14:12] } + { 1'b0, b[14:12] } + { 3'b000, cout11 };

        data_out[15] = (a[15] ^ b[15]) ^ cin15;
        cout15 = (cin15 & (a[15] ^ b[15])) | (a[15] & b[15]);
    end

    // ...
```

Logische, Shift- und Rotations- sowie Bit-Set/Reset Operationen sind mit individuellen, weitestgehend selbsterklärenden Verilog-Statements implementiert.

Die ALU implementiert weiterhin die `DAA` Operation. Diese ist dabei prinzipiell durch eine Reihe von Fallunterscheidungen realisiert, welche in Ahängigkeit der eingehenden Flags und des Wertebereiches des als Operand A angelegten Akkumulators-Inhaltes dessen korrigierten Wert bestimmen. Die `DAA`-Instruktion kann dabei in einem Takt abgearbeitet werden. alle möglichen Korrekturfaktoren sind als Konstanten im Source-Code der ALU spezifiziert.

Nehmen Akkumulator und Flag-Register vor Ausführung der `DAA` Instruktion Werte an, die im Kontext von `BCD`-Rechnungen so nicht auftreten sollten, ist das Ergebnis laut Dokumentation undefiniert. Wir haben uns entschieden das Verhalten der `DAA` Operation in diesem Fall exakt dem der `DAA` Instruktion anzupassen, die im von uns während Verifikation verwendeten `z80sim` Software-Simulator implementiert ist. Die Hardware-Beschreibung ist hierdurch komplizierter als unbedingt notwendig, aber dafür wird die automatisierte Test-Generierung erleichtert (siehe Abschnitt 5).

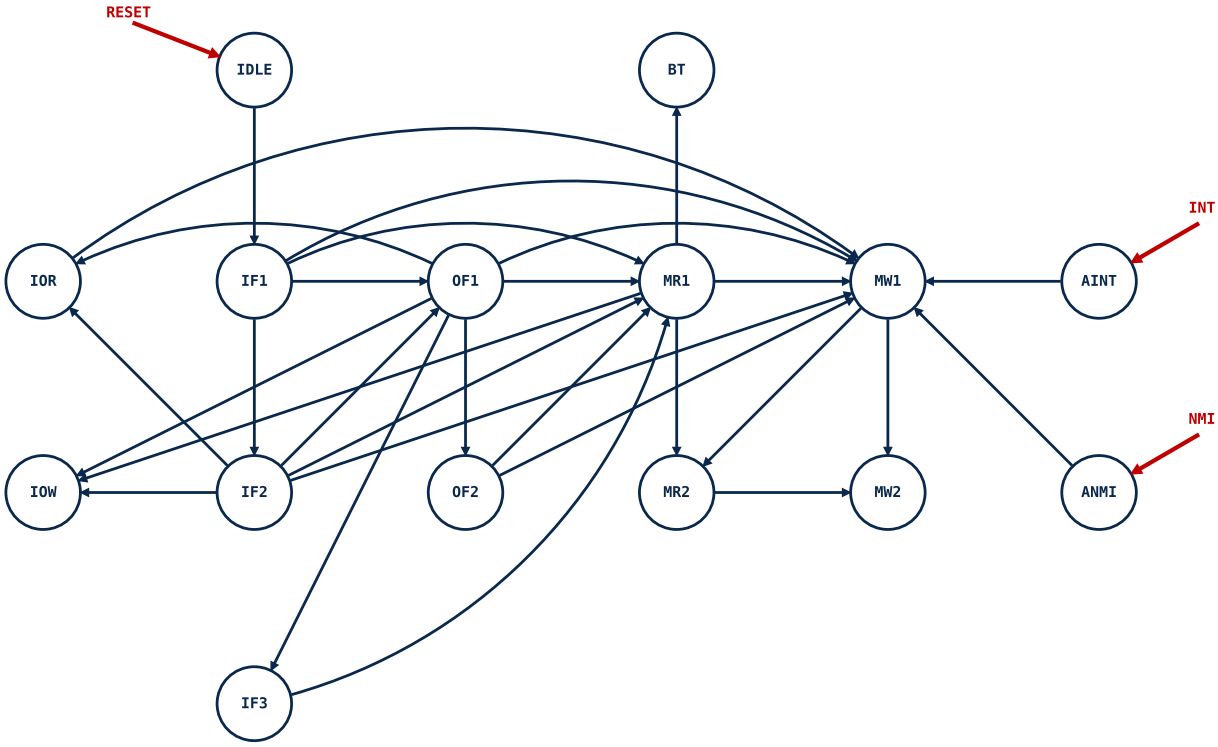

Abbildung 3.6: Vereinfachter Zustandsübergangsgraph

3.1.2 CONTROLLER

Der Controller kapselt FSM und Decoder. In diese beiden Module ist der größte Entwicklungsaufwand geflossen und ihre große Komplexität rechtfertigt ihre Trennung.

3.1.2.1 FSM

Die Aufgabe der Finite State Machine besteht in der Koordinierung des Ablaufs der Befehlsausführung. Der aktuelle Zustand wird intern in einem Register gespeichert und über den Ausgang `state` dem Decoder zur Verfügung gestellt. Der Folgezustand wird kombinatorisch in Abhängigkeit vom aktuellen Zustand, Opcode-Präfix und Opcode bestimmt und mit der nächsten positiven Taktflanke übernommen. Abbildung 3.6 zeigt den vereinfachten Zustandsübergangsgraphen der FSM, in dem für jeden möglichen Zustandsübergang eine Kante zwischen zwei Zuständen eingezeichnet ist. Dabei werden folgende Abkürzungen genutzt:

IF:	Instruction Fetch
OF:	Operand Fetch
MR:	Memory Read
MW:	Memory Write
IOR:	IO Read
IOW:	IO Write
BT:	Block Transfer
AINT:	Interrupt Acknowledge
ANMI:	Non-maskable Interrupt Acknowledge

Die gezeigten Zustände entsprechen den Maschinen-Zyklen der Implementierung, die wiederum mehrere Takte andauern können und demnach aus mehreren Unterzuständen zusammengesetzt sind. Die möglichen Zustandsübergänge zurück zu `IF1` sind der Übersichtlichkeit halber nicht dargestellt.

Nach einem initialen Reset befindet sich die FSM im *Idle* Zustand und verbleibt solange in diesem, bis der gesamte Programmspeicher von einer externen Quelle eingelesen worden ist[3]. Danach wechselt die FSM in den ersten *Instruction Fetch* Zustand (`IF1`), der intern aus drei Unterzuständen [4] zusammengesetzt ist. Ist das in diesem Zustand eingelesene Byte ein Opcode-Präfix (oder ein Teil eines solchen) folgt ein weiterer `Instruction Fetch` Zustand (`IF2`).

Anschließend erfolgt dann eine Reihe von Übergängen zwischen den übrigen Zuständen, jeweils durch den über die Eingänge `reg_instr` und `reg_op_prefix` innerhalb der FSM verfügbaren Opcode und Opcode-Präfix bestimmt. Listing 3.3 zeigt einen relevanten Code-Ausschnitt. Mittels des `casez` Statements können in Opcodes kodierte Register o.ä. lesbar zusammengefasst werden. Für die Opcodes wurden keine Mnemoniken eingeführt, da gleiche Opcodes in Abhängigkeit vom Opcode-Präfix komplett unterschiedliche Bedeutungen annehmen können.

Listing 3.3: FSM-Zustandübergangslogik (Ausschnitt `fsm.v`)

```
case(state)

    // ...

    FSM_STATE_INSTR_FETCH2_3: begin
        casez ({ reg_op_prefix, reg_instr })
            { OP_PREFIX_DD, 8'b00100001 }, // LD IX,nn
            { OP_PREFIX_DD, 8'b00100010 }, // LD (nn),IX
            { OP_PREFIX_DD, 8'b00101010 }, // LD IX,(nn)
            { OP_PREFIX_DD, 8'b00110100 }, // INC (IX + d)
            { OP_PREFIX_DD, 8'b00110101 }, // DEC (IX + d)
            { OP_PREFIX_DD, 8'b00110110 }, // LD (IX + d),n
            { OP_PREFIX_DD, 8'b01110??? }, // LD (IX + d),r
            { OP_PREFIX_DD, 8'b01???110 }, // LD r,(IX + d)
            { OP_PREFIX_DD, 8'b10???110 }, // ALUOP [A,](IX + d)
            { OP_PREFIX_DD, 8'b11001011 }, // (ROT|SHIFT|BIT) (IX + d)
            { OP_PREFIX_ED, 8'b01??0011 }, // LD (nn),dd
            { OP_PREFIX_ED, 8'b01??1011 }, // LD dd,(nn)
            { OP_PREFIX_FD, 8'b00100001 }, // LD IY,nn
            { OP_PREFIX_FD, 8'b00100010 }, // LD (nn),IY
            { OP_PREFIX_FD, 8'b00101010 }, // LD IY,(nn)
            { OP_PREFIX_FD, 8'b00110100 }, // INC (IY + d)
            { OP_PREFIX_FD, 8'b00110101 }, // DEC (IY + d)
            { OP_PREFIX_FD, 8'b00110110 }, // LD (IY + d),n
            { OP_PREFIX_FD, 8'b01110??? }, // LD (IY + d),r
            { OP_PREFIX_FD, 8'b01???110 }, // LD r,(IY + d)
            { OP_PREFIX_FD, 8'b10???110 }, // ALUOP [A,](IY + d)
            { OP_PREFIX_FD, 8'b11001011 }: // (ROT|SHIFT|BIT) (IY + d)
            begin
                state_next   = FSM_STATE_OP_FETCH1_1;
                int_suppress = 1'b1;
            end

            // ...

            default: begin
                state_next = FSM_STATE_INSTR_FETCH1_1;
            end

    // ...
```

[3] Siehe Abschnitt 3.2.1

[4] Im Verilog-Code in diesem Fall durch die Konstanten `FSM_STATE_INSTR_FETCH1_1`, `FSM_STATE_INSTR_FETCH1_1_2` und `FSM_STATE_INSTR_FETCH1_3` identifiziert.

Besonders hevorzuheben ist der Block Transfer Zustand (BT). Dieser wird nur während der CPIR und CPDR Instruktionen benötigt und ist außer dem IDLE Zustand der einzige, in dem kein Speicher- oder IO Zugriff erfolgt. Dies ist die einzige Ausnahme zum in Abschnitt 2 beschriebenen Prinzip der minimalen Intruktions-Länge.

Die Interrupt Acknowledge AINT und Non-maskable Interrupt Acknowledge Zustände (AINT und ANMI) werden nicht während der normalen Befehlsabarbeitung, sondern erst nach Auftreten entsprechender Interrupts erreicht. Die FSM übernimmt die Verwaltung des aktuellen Interrupt-Zustandes und tastet Interrupt-Anfragen wie in der Dokumentation beschrieben mit der steigenden Flanke des letzten Taktes jeder Instruktion ab.

Die Interrupt Control Flipflops int_iff1 und int_iff2 sind dabei ebenfalls Teil der FSM und werden, wenn die Eingänge int_enable bzw. int_disable aktiv sind, mit der nächsten steigenden Taktflanke gesetzt bzw. zurückgesetzt. Das Umschalten des Interrupt-Modus sowie das Wiederherstellen von int_iff1 nach Ende eines nicht maskierbaren Interrupts sind ebenfalls in der FSM implementiert.

Im vorletzten Takt jeder Instruktion wird das interne int_sample Signal gesetzt. Ist dieses aktiv, werden mit der folgenden steigenden Flanke des letzten Taktes der Instruktion die über int_request_int an der FSM anliegenden Interrupt Requests gesampelt.

Ein nicht maskierbarer Interrupt Request hat dabei Priorität und hat das Setzen des int_sampled_nmi Flipflops zur Folge. Für maskierbare Interrupts wird das entsprechende int_sampled_int Flipflop nur dann gesetzt, wenn auch das int_iff1 Flipflop gesetzt ist und im letzten Takt kein Interrupt-Enable stattgefunden hat[5].

Einen Sonderfall bilden die Instruction Fetches. Für diese ist erst nach dem Einlesen des Bytes aus dem Programmspeicher entscheidbar, ob der Instruction Fetch Maschinen-Zyklus der letzte der aktuellen Instruktion ist. Da das Einlesen dieses Bytes mit dem Samplen der Interrupt Requests zusammenfällt, musste hier das zusätzliche Signal int_suppress eingeführt werden. Bei Instruction Fetches werden die Interrupt Request immer gesampelt und notfalls mit int_suppress das Abarbeiten eines Interrupt-Requests mitten in einer Instruktion unterdrückt.

Nach Auftreten eines Interrupts wechselt die FSM in den Interrupt-Acknowledge oder den Non-maskable Interrupt Acknowledge Zustand und beginnt mit der in der Dokumentation beschriebenen Interrupt-Behandlung. Währenddessen ist eines der Signale int_active_int oder int_active_nmi aktiv, über die FSM und Decoder z.B. im Memory Write Zustand, der Teil beider Interrupt Acknowledge Vorgänge ist, erkennen können, dass spezielle Operationen durchzuführen sind.

Der Interrupt Modus ist dabei im Register int_mode in der FSM gespeichert und wird an den Decoder weitergereicht. Interrupt Mode 0 ist nicht vollständig implementiert, das Ausführen Instruktionen beliebiger Länge hätte einen starken Eingriff in das bestehende Design erfordert. Außerdem gibt es wenige realistischen Anwendungsfälle der Schaltung in denen Interrupt-Mode 0 nützlich wäre. Unsere Implementierung kann im Interrupt Modus 0 daher nur ein-Byte Instruktionen ausführen.

3.1.2.2 DECODER

Der Decoder ist ein rein kombinatorisches Modul, welches in Abhängigkeit vom aktuellen Zustand, der eingelesenen Instruktion sowie gesetzten Flags und dem Interrupt-Zustand Kontrollsignale für alle Datenpfad-Elemente sowie Speicher- und IO-Zugriffe generiert.

Zentrales Element der decoder.v Source-Datei ist ein case-Statement, dessen Struktur die des im FSM Code-Ausschnitt in Listing 3.3 gezeigten case-Statements widerspiegelt.

Um Übersichtlichkeit zu wahren und dem Synthese-Compiler die Eliminierung gemeinsamer Teilausdrücke zu erleichtern werden einige in mehreren Zuständen identisch bestimmte Signale

[5] Über den int_enable_buf Flipflop realisiert.

in einem separaten kombinatorischen `always`-Block generiert. Dies betrifft Branch-Bedingungen (beispielhaft in Listing 3.4 gezeigt) und im Opcode kodierte Register und ALU-Modi.

Listing 3.4: Dekodierung Branch-Bedingung (Ausschnitt `decoder.v`)

```
reg cond

// ...

case (reg_instr[5:3])
    3'b000: cond = !regfile_flag_z;
    3'b001: cond =  regfile_flag_z;
    3'b010: cond = !regfile_flag_c;
    3'b011: cond =  regfile_flag_c;
    3'b100: cond = !regfile_flag_pv;
    3'b101: cond =  regfile_flag_pv;
    3'b110: cond = !regfile_flag_s;
    3'b111: cond =  regfile_flag_s;
endcase
```

Fast alle Maschinen-Zyklen sind so strukturiert, dass erst in ihrem letzten Takt in Abhängigkeit vom aktuellen Prozessorzustand Steuersignale generiert werden, die die für die korrekte Ausführung der Instruktion nötigen Datenoperationen in Bewegung setzen. In den vorangehenden Takten werden meist nur Kontrollsignale für den Speicher- oder IO-Zugriff erzeugt, der mit wenigen Ausnahmen in jedem Maschinen-Zyklus stattfindet. So werden zum Beispiel im ersten Takt jedes Instruction Fetch nur Steuersignale für den Programmspeicher erzeugt, im zweiten zusätzlich der PC inkrementiert und `reg_instr` oder `reg_op_prefix` beschrieben. Erst im dritten Takt werden je nach eingelesenem Byte z.B. spezifische ALU-Operationen ausgeführt. Der relevante Verilog-Code ist in Listing 3.5 gezeigt.

Listing 3.5: Struktur der Instruction Fetch Steuersignal-Generierung (Ausschnitt `decoder.v`)

```
case (state)
    FSM_STATE_INSTR_FETCH1_1,
    FSM_STATE_INSTR_FETCH2_1,
    FSM_STATE_INSTR_FETCH3_1:: begin
        mem_n_m1         = 1'b0;
        mem_n_pmem       = 1'b0;
        mem_n_mreq       = 1'b0;
        mem_n_rd         = 1'b0;
        mux_mem_addr_sel = MUX_MEM_ADDR_SEL_REG_PC;
    end
    FSM_STATE_INSTR_FETCH1_2,
    FSM_STATE_INSTR_FETCH2_2,
    FSM_STATE_INSTR_FETCH3_2: begin
        mem_n_m1         = 1'b0;
        mem_n_pmem       = 1'b0;
        mem_n_mreq       = 1'b0;
        mem_n_rd         = 1'b0;
        mux_mem_addr_sel = MUX_MEM_ADDR_SEL_REG_PC;

        alu_mode         = ALU_MODE_INC;
        mux_alu_op_a_sel = MUX_ALU_OP_A_SEL_REG_PC;
        mux_int_bus_sel  = MUX_INT_BUS_SEL_ALU_OUT;
        reg_pc_we        = 1'b1;

        reg_instr_we = 1'b1;
    end
    FSM_STATE_INSTR_FETCH1_3: begin

        // ...
```

Ähnliches gilt für andere Maschinen-Zyklen. Dieses Vorgehen bietet sich an, da z.B. für Maschinen-Zyklen, in denen ein Byte eingelesen wird, in den meisten Fällen sowieso erst dann

eine sinnvolle nächste Datenoperation durchgeführt werden kann, wenn dieses Byte verfügbar ist, d.h. im letzten Takt des Maschinen-Zyklus.

Die Speicher- und IO Steuersignale müssen in fast allen Fällen über mehrere Takte eines Maschinen-Zyklus gehalten werden. Speicher-Adresse, Ausgangdaten und einige andere Signale werden dabei immer bereits im letzten Takt des vorhergehenden Maschinen-Zyklus generiert um das in Abschnitt 4 beschriebene Timing-Verhalten zu erzeugen.

3.2 SPEICHERVERWALTUNG

Die Hauptaufgabe der Speicherverwaltung ist es, eine Abstraktionsschicht für Speicherzugriffe zu schaffen, deren nach außen (d.h. zur CPU hin) sichtbares Verhalten im Einklang mit dem in der Dokumentation beschriebenen Speichermodell ist (d.h. ein kontinuierlicher 16-Bit Adressraum in dem sowohl Programme als auch Daten abgelegt sind).

Dadurch wird es unter anderem möglich unabhängig von der Implementierung der CPU Änderungen an der konkreten Speicher-Architektur vorzunehmen. Insbesondere existiert so die Trennung in externen Speicher und internen Programm- bzw. Datenspeicher aus Sicht der CPU nicht.

Das `memory_control` Modul kapselt dabei die `ipram` und `idram` Module, welche wiederum dünne Wrapper um den verwendeten internen Programm- und Datenspeicher sind.

Die Eingänge der Speicherverwaltung sind die von der CPU erzeugten Speicher-Kontrollsignale, die alle ein externes Pendant haben [6].

Einige dieser Signale können einfach zum Toplevel hin durchgereicht werden. Für $\overline{\texttt{mreq}}$, $\overline{\texttt{rd}}$ und $\overline{\texttt{wr}}$ passiert dies jedoch nur dann, wenn auch tatsächlich ein externer Speicherzugriff erforderlich ist, was anhand der anliegenden Speicher-Adresse entschieden werden kann. Nach dem gleichen Prinzip wird für lesende Operationen der korrekte Datenbus auf den `int_dout` Ausgang der Speicherverwaltung geschaltet, der mit dem Dateneingang der CPU verbunden ist. Listing 3.6 zeigt einen entsprechenden Code-Ausschnitt für Zugriffe auf den Programmspeicher.

Listing 3.6: Programmspeicherzugriff (Ausschnitt `memory_control.v`)

```
// ...

if (!int_n_pmem) begin
    if (!int_n_rd && int_n_wr) begin
        if (int_addr < INTERNAL_PROGMEM_SIZE) begin
            ipram_ce = 1'b1;

            int_dout = ipram_dout;
        end
    end
    else begin
        ext_n_mreq = 1'b0;
        ext_n_rd   = 1'b0;

        int_dout = ext_din;
    end
end

// ...
```

[6] Mit Ausnahme des `int_n_pmem` Signals, das aufgrund der internen Trennung von Programm- und Datenspeicher notwendig ist um zwischen Zugriffen auf diese zu unterscheiden.

3.2.1 INTERNER PROGRAMMSPEICHER

Das `ipram` Modul ist wiederum eine Abstraktionsschicht für den verwendeten `SY180_1024X8X4CM4` Speicher (im Folgenden mit *Speicher-Element* bezeichnet).

Nach jedem Reset des Chips wird der gesamte interne Programmspeicher aus einer externen Quelle eingelesen. Im `ipram` Modul existiert zu diesem Zweck ein 13-Bit Zähler, der bei einem Reset auf null gesetzt und anschließend mit jeder steigenden Tatkflanke inkrementiert wird.

Solange die Initialisierung noch nicht abgeschlossen ist, ist auch das interne `loaded` Signal nicht aktiv und Speicher-Element wird Takt für Takt und Adresse für Adresse (von null beginnend) mit den jeweils am `din` Eingang anliegenden Bytes gefüllt. Währenddessen ist der Chip-Enable Eingang des Speicher-Elementes ($\overline{\texttt{csb}}$) kontinuierlich aktiv.

Schreibzugriffe auf das Speicher-Element erfolgen auf Basis von 32-Bit Worten. Vier separate low-aktive write enable Eingänge (`bwe`) kontrollieren welche der vier Bytes des am Schreibeingang des Speicher-Elementes anliegenden 32-Bit Wortes bei steigender Taktflanke tatsächlich übernommen werden. Einzelne Bytes können somit in den Programmspeicher geschrieben werden, indem das am Dateneingang des `ipram` Moduls anliegende Byte parallel an jedes der vier Bytes dieses Schreibeinganges angelegt wird und die unteren beiden Bits des Initialisierungs-Zählers genutzt werden um zu bestimmen, welches dieser Bytes in einem gegebenen Takt geschrieben wird. Die verbleibenden Bits des Zählers wählen das zu beschreibende 32-Bit Wort aus (`addr_word`).

Erreicht der Initialisierungs-Zähler den Wert `13'b1000000000000` ist das Speicher-Element vollständig geladen und das `loaded` Signal wird aktiviert. Anschließend ist der Programmspeicher nur noch für lesende Zugriffe verfügbar. Für diese wird auf ähnliche Weise anhand der unteren beiden Bits der anliegenden Adresse entschieden, welches Byte des am Ausgang des Speicher-Elementes anliegenden 32-Bit Wortes (`dout_int`) vom `ipram` Modul ausgegeben werden muss.

Listing 3.7: Programmspeicher-Initialisierung (Ausschnitt `memory_control.v`)

```
// ...

assign loaded = (counter == 13'b1000000000000);

always @(posedge clk, negedge n_reset) begin
    if (n_reset == 1'b0) begin
        counter <= 13'b0000000000000;
    end
    else if (!loaded) begin
        counter <= counter + 13'b0000000000001;
    end
end

always @* begin
    #10

    din_int = din;
    dout    = 8'h00;
    bwe     = 4'b1111;

    if (!loaded) begin
        csb = 1'b0;

        addr_word = counter[11:2];

        case (counter[1:0])
            2'b00: bwe[3] = 1'b0;
            2'b01: bwe[2] = 1'b0;
            2'b10: bwe[1] = 1'b0;
            2'b11: bwe[0] = 1'b0;
        endcase
    end
    else begin
        csb = ~ce;

        addr_word = addr[11:2];

        case (addr[1:0])
            2'b00: dout = dout_int[31:24];
            2'b01: dout = dout_int[23:16];
            2'b10: dout = dout_int[15:8];
            2'b11: dout = dout_int[7:0];
        endcase
    end
end
```

Listing 3.7 zeigt den Verilog-Code, der das beschriebene Verhalten implementiert. Die Verzögerung zu Beginn der zweiten `always`-Schleife dient dazu, Timing-Warnungen im Simulator-Output zu unterdrücken, die für das Synthese-Ergebnis nicht relevant sind.

Das `loaded` Signal wird außerdem von der Memory Control an die CPU weitergereicht und signalisiert dieser, dass mit der Ausführung des geladenen Programms begonnen werden kann.

3.2.2 INTERNER DATENSPEICHER

Der Aufbau des `idram` Moduls ist deutlich weniger kompliziert. Intern sind in diesem zwei separate `SY180_256X8X1CM8` Speicher-Elemente instanziiert. Anhand der angelegten Adresse wird entschieden, welches dieser beiden Speicher-Elemente aktiviert werden muss. Die unteren 2^8 wählbaren Adressen sind einem der Elemente zugeordnet, die restlichen 2^8 Adressen dem anderen.

3.3 TOPLEVEL UND PADS

Das Modul z80 kapselt die Module cpu und memory_control. Die Pads sind in einem eigenen Modul (pads) implementiert und dieses ist im top_z80 Modul zusammen mit dem z80 Modul instanziiert.

Die Ausgänge des z80 Moduls werden direkt durch Register getrieben um die Stabilität dieser Signale zu garantieren. Die Werte der internen Speicher- und IO Steursignale und Daten werden dabei zu jeder positiven Taktflanke automatisch übernommen und liegen somit um einen Takt verzögert über die Pads am Ausgang des Chips an.

Im pad Modul ist die in Listing 3.8 gezeigte, zum externen Datenbus gehörige, Pad-Instanz hervorzuheben. Da sich ausgehende und eingehende Daten-Bytes den gleichen externen Bus teilen, intern aber über die getrennten Busse data_out und data_in von und zum Datenpfad laufen, musste hier das (vom Decoder generierte) data_out_en Kontrollsignal eingeführt werden. Ist dieses aktiv, wird das interne data_out Signal auf den externen Datenbus ausgegeben, andernfalls nimmt das interne data_in Signal den Wert des externen Datenbus an.

Listing 3.8: IO-Pad Instanz für den externen Datenbus (Ausschnitt pads.v)

```
ZMA2GSC data_pad_i [7:0] (
    .IO(IO_DATA),
    .I(data_out),
    .O(data_in),
    .E(data_out_en),
    .E2(1'b1),
    .E4(1'b1),
    .E8(1'b0),
    .SR(1'b1),
    .SMT(1'b0),
    .PU(1'b1),
    .PD(1'b1)
);
```

4 TIMING

Im Folgenden wird anhand einiger repräsentativer Waveforms das von unserer Implementierung gezeigte Timing-Verhalten relevanter Steuersignale bei internen und externen Speicher- und Gerätezugriffen sowie Interrupt-Ackknowledgements und nach `HALT` Instruktionen aufgezeigt. Dieses unterscheidet sich im Wesentlich durch die Anforderung, alle Steuersignale nur an positiven Taktflanken umzuschalten, von der Dokumentation.

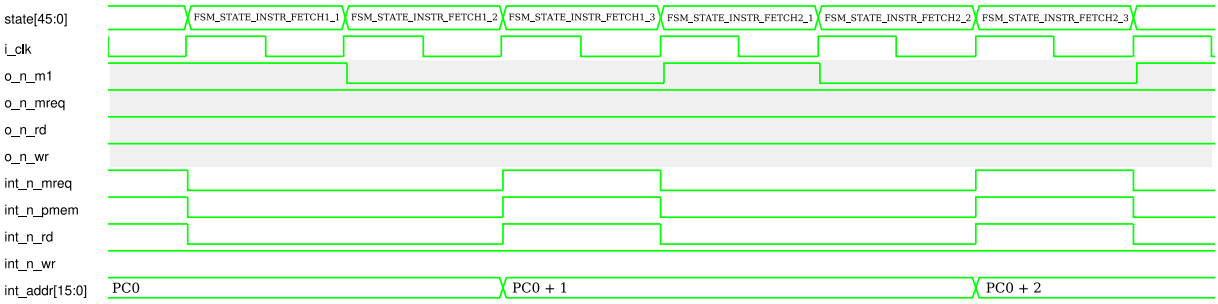

Abbildung 4.1: Interner lesender Programmspeicher-Zugriff

In Abbildung 4.1 ist das Steuersignal-Timing für das Auslesen zweier Bytes aus dem internen Programmspeicher dargestellt. Die `int*`-Signale sind Eingänge des `memory_control` Moduls, welches -unter anderem in Abhängigkeit der angelegten Speicher-Adresse- die passenden Kontrollsignale für den internen Programmspeicher generiert oder alternativ die Anfrage durch Erzeugung passender externer Kontrollsignale an ein off-chip Speicherelement weiterleitet. Letztere Signale sind (grau unterlegt) ebenfalls dargestellt.

Zu erkennen ist, dass die internen Steuersignale jeweils in den ersten zwei Takten der drei Takte andauernden Leseoperationen aktiv sind. Dies ist unbedingt notwendig, denn die entsprechenden externen Signale können nur mit einer Verzögerung von einem Takt generiert werden, da diese direkt von synchronen Registern im `z80` Modul getrieben werden, gleichzeitig aber bei internen und externen Speicherzugriffen das eingelesene Byte an der gleichen Taktflanke abgetastet werden muss[1]. Andernfalls müssten interne Komponenten der CPU wie Decoder und FSM ihr Verhalten abhängig davon, ob aus dem internen oder externen Speicher gelesen werden soll, dynamisch anpassen. Idealerweise sollte diesen Komponenten aber die Existenz eines einzigen kontinuierlichen Adressraums vorgetäuscht werden, um sie robust gegenüber Änderungen an der Speicher-Infrastruktur zu machen.

Die Speicheradresse wird bereits im jeweils letzten Takt des **vorherigen** Maschinen-Zyklus angelegt, damit diese stabil ist, wenn `int_n_mreq` bzw. `o_n_mreq` aktiv wird. Die Adresse wird solange gehalten, bis das einzulesende Byte abgetastet worden ist.

Zu beachten ist weiterhin das `o_n_m1` Signal, welches in Übereinstimmung mit der Dokumentation bei jedem Instruction Fetch (hier zwei in Folge) aktiv wird, auch dann, wenn der eigentliche Zugriff auf den internen Speicher erfolgt. Außerdem existiert das zusätzliche `int_n_pmem` Steuersignal, welches durch die interne Trennung von Programm- und Datenspeicher notwendig wird und dem `memory_control` Modul bei internem Speicherzugriff signalisiert, welcher von beiden angesprochen werden muss.

[1] Hier die Flanke beim Übergang vom jeweiligen zweiten zum dritten Takt.

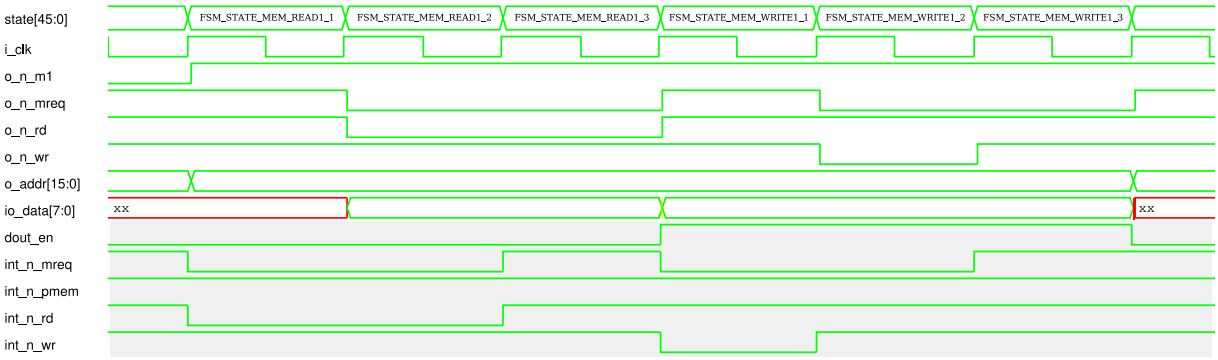

Abbildung 4.2: Externer lesender und schreibender Datenspeicher-Zugriff

Abbildung 4.2 zeigt das Steuersignal-Timing für einen Lesezugriff gefolgt von einem Schreibzugriff auf den externen Speicher. Hier ist nocheinmal die Verzögerung der externen gegenüber den internen Signalen (hier grau unterlegt) verdeutlicht. Zusätzlich ist hier das Signal `dout_en` wichtig, welches durch Ansteuerung des entsprechenden IO-Pins das zu schreibende Byte auf den externen tristate Datenbus ausgibt. Dieses wird zusammen mit `dout_en` analog zur Speicheradresse intern bereits im jeweiligen Zustand vor `FSM_STATE_MEM_READ*_1` angelegt. Außerdem ist zu beachten, dass das `int_n_wr` bzw. `o_n_wr` Signal beim schreibenden Speicherzugriff nur in einem Takt aktiv ist, sodass Adresse und Daten in Übereinstimmung mit der Dokumentation noch einen Takt nach Deaktivierung von `n_wr` stabil sind.

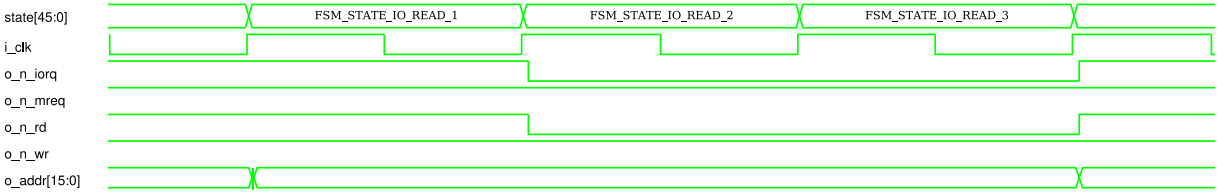

Abbildung 4.3: IO Lesezugriff

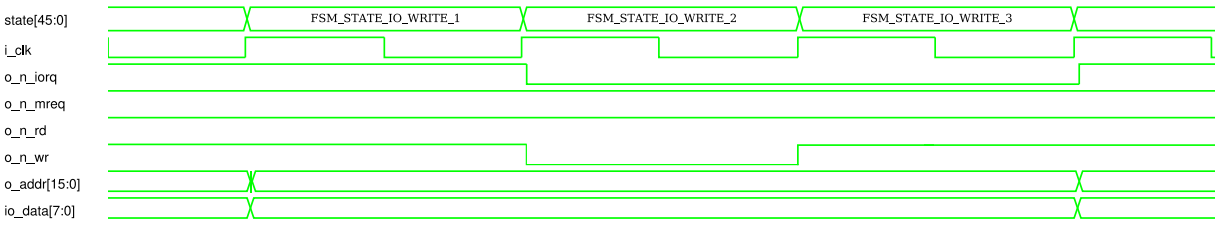

Abbildung 4.4: IO Schreibzugriff

Abbildung 4.3 und Abbildung 4.4 zeigen die relevanten Signalverläufe für lesende bzw. schreibende Zugriffe auf externe Geräte. Diese sind selbsterklärend, `o_n_iorq` ersetzt hier `o_n_mreq`.

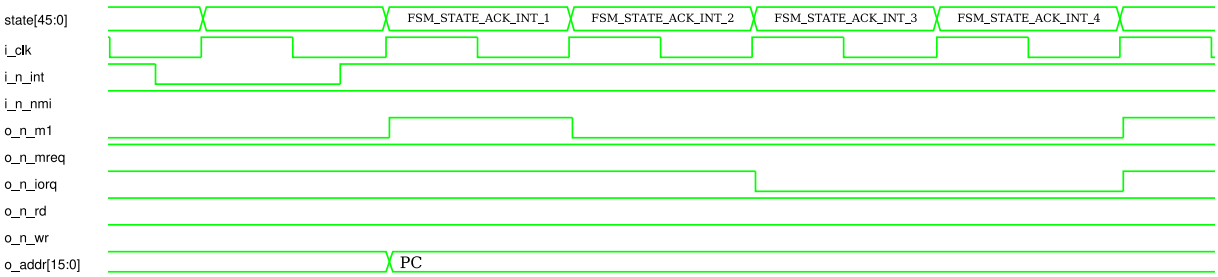

Abbildung 4.5: Interrupt Acknowledgement

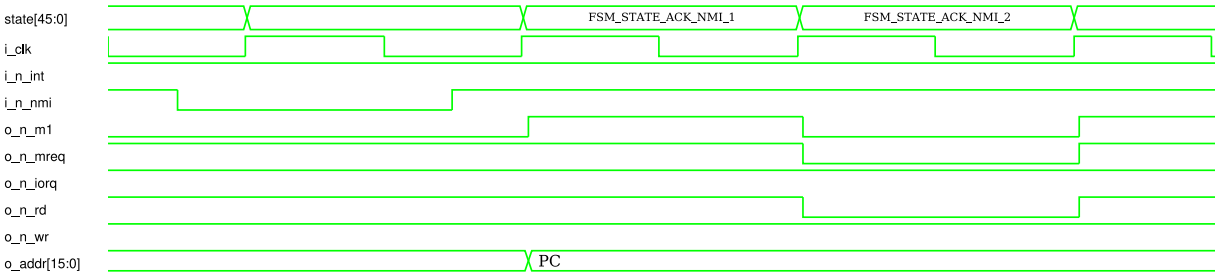

Abbildung 4.6: Non-maskable Interrupt Acknowledgement

Abbildung 4.5 und Abbildung 4.6 zeigen die Signalverläufe während der Interrupt Acknowledgements. Diese orientieren sich an der Dokumentation. Speicher-Operationen wie das Ablegen des Program Counters auf dem Stack, welche im Zuge eines Interrupt Acknowledgements durchgeführt werden müssen[2], werden im Anschluss an die hier gezeigten Zustände in gewöhnlichen `FSM_STATE_MEM_WRITE*` etc. Zuständen durchgeführt.

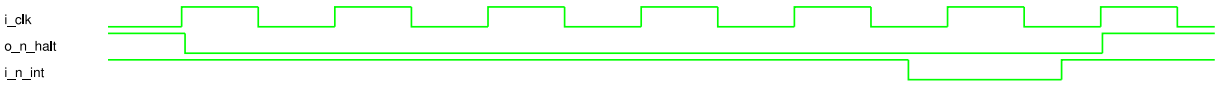

Abbildung 4.7: HALT

Abbildung 4.7 zeigt den Effekt einer `HALT` Instruktion. Es werden kontinuierlich No-Ops ausgeführt -wobei der `o_n_halt` Ausgang aktiv ist- bis ein Interrupt (maskierbar oder nicht-maskierbar) auftritt und abgehandelt wird.

[2] Im Falle eines maskierbare Interrupts in Abhängigkeit vom eingestellten Interrupt-Modus.

5 VERIFIKATION

Einer der wichtigsten Schritte bei der Entwicklung einer komplexen digitalen Schaltung ist die Verifikation von deren Korrektheit. Im Umfang dieser Belegarbeit ist das letztendliche Ziel sicherzustellen, dass das simulierte Verhalten der entworfenen Schaltung nach abgeschlossenem Layout-Schritt mit dem dokumentierten Verhalten eines Z80 Prozessors übereinstimmt.

Im Folgenden werden die dabei von uns verfolgten Grundansätze und die zur automatischen Testgenerierung und -ausführung verwendeten Methoden beschrieben. Dabei wird immer wieder der Begriff *Testfall* verwendet, der hier eine `testcase.v` Datei im Kontext der von uns entwickelten Testumgebung bezeichnet. Ein Testfall setzt sich dabei stets aus mehreren *Tests* zusammen, wobei jeder Test die Funktionalität einer Instruktion für einen bestimmten Ausgangs-Systemzustand verifiziert.

5.1 GRUNDANSÄTZE

Bei der Entwicklung des Prozessors haben wir uns an dem in der Software-Entwicklung weit verbreiteten Prinzip der *Testgetriebenen Entwicklung* (im Folgenden kurz `TDD`) orientiert.

Nachdem mit dem Entwurf eines Minimal-Prozessors eine grobe Modulstruktur etabliert war, wurde bei der schrittweisen Implementierung jeder weiteren Instruktion jeweils **bevor** die notwendigen Änderungen an FSM, Decoder (und in manchen Fällen auch am Datenpfad) durchgeführt wurden, ein Testfall angelegt, dessen Bestehen ein ausreichend guter Indikator für die korrekte Funktionalität der neu hinzukommenden Instruktion ist. Anschließend wurde der Verilog-Code jeweils bis zum Bestehen aller Tests in diesem Testfall modifiziert. Erst danach wurden die entsprechenden Änderungen in die Versionsverwaltung übernommen.

Dieses Vorgehen wurde strikt für alle implementierten Instruktionen eingehalten. So konnte gewährleistet werden, dass zu keinem Zeitpunkt fehlerhafte oder unvollständige Instruktionen in das Design übernommen wurden. Zudem wurden in periodischem Abstand automatisiert alle bisher entwickelten Test ausgeführt um durch Abhängigkeiten zwischen verschiedenen Instruktionen oder Änderungen an von mehreren Instruktionen genutzten Elementen (wie zum Beispiel dem Speichercontroller) entstandene Fehler rechtzeitig erkennen und beheben zu können.

Wir haben uns in diesem Zusammenhang auch dafür entschieden, letztendlich nur das nach außen für einen Anwender der resultierenden Schaltung sichtbare Verhalten (welches bei korrekter Implementierung deckungsgleich mit dem in der Dokumentation beschriebenen sein sollte) zu testen. Das Verhalten von Untermodulen wie beispielsweise der ALU oder der FSM wurde bei deren initialer Entwicklung durch eigene, speziell auf diese Komponenten abgerichtete Tests überprüft. Im Verlauf des Projektes wurden alle derartigen Tests aber zugunsten von Toplevel Tests, welche lediglich durch Instruktionen oder äußere Stimuli hervorgerufene System-Übergänge testen, wieder entfernt. Der Gedanke hinter dieser Entscheidung ist, dass das Verhalten der internen Komponenten letztlich irrelevant ist, solange es nach außen hin den gewünschten Effekt hat. Dieses nicht zu testen spart dabei wertvolle Entwicklungszeit und erhöht die "Stabilität" der Tests gegenüber internen Strukturänderungen. Auch die genaue Ausführungszeit einzelner Instruktionen in Hinsicht auf Maschinen-Zyklen und T-states wird nicht getestet, da diese in den meisten Fällen in unserer Implementierung ohnehin gewollt von der dokumentierten abweichen.

Dieses Vorgehen ist gemeinhin als `Black-Box-Testing` bekannt. Einzig problematisch ist hier-

bei eventuelle fehlerhafte oder überflüssige Funktionalität in Untermodulen, die sich nicht bis zum Toplevel fortpflanzt und bei der angewandten Testmethodik somit unbemerkt bleibt. Durch Analyse der durch die Tests erzeugten Code Coverage kann dieser Fall aber größtenteils ausgeschlossen werden.

Ein weiteres Problem, vor das wir uns in Bezug auf die Test-Entwicklung bereits bei der Entwicklung des Minimal-Prozessors gestellt sahen, ist die durch die komplexe Befehlssatzarchitektur des Z80 bedingte riesige Menge an Tests, die nötig sind um die korrekte Funktionalität aller Instruktionen in allen erdenklichen Grenzfällen sicherzustellen. Eine erschöpfende Verifikation ist von vornherein ausgeschlossen, da hierzu das Verhalten jeder Instruktion jeweils für alle möglichen Prozessor- und Speicher-Ausgangszustände sichergestellt werden müsste. Aber auch eine annehmbare Testabdeckung, welche für jede Instruktion jeden relevanten Fall (z.B. das korrekte Setzen aller von der Instruktion beeinflussten Flags) zumindest einmalig testet, ist ohne automatisierte Erzeugung der Testfälle fast nicht erreichbar. Bei mehreren hundert möglichen Instruktionen sollten schätzungsweise zumindest einige tausende Tests nötig sein um eine derartige Testabdeckung zu erreichen. Das Schreiben der Tests würde somit den Großteil der gesamten zur Verfügung stehenden Entwicklungszeit beanspruchen, vor allem auch da so zusätzlich äußerste Sorgfalt nötig ist um sicherzustellen, dass die Testfälle sowohl vollständig als auch korrekt sind.

In diesem Zusammenhang steigt dann auch die Wahrscheinlichkeit, dass sich durch menschliches Versagen Fehler in die Tests selbst einschleichen. Dies führt wiederum zu erhöhtem Entwicklungsaufwand, da hierdurch bei der Diagnose der Ursache fehlschlagender Tests sowohl Fehler in der Implementierung als auch Fehler in den Tests selbst in Betracht gezogen werden müssen. Außerdem kann es passieren, dass Fehler in der Implementierung unentdeckt bleiben, wenn fehlerhafte Tests die gewünschte Verhaltensprüfung nicht oder nur teilweise implementieren. Im schlimmsten Fall kann es auch dazu kommen, dass das im Test geprüfte Verhalten zum Beispiel durch ein Missverständnis seitens des Entwicklers zwar mit dem Verhalten der Implementierung, aber nicht mit dem in der Dokumentation beschriebenen Verhalten übereinstimmt. Derartige Fehler in der Implementierung bleiben dann mit großer Wahrscheinlichkeit ebenfalls unentdeckt. Zudem wird dem Entwickler durch das Bestehen der zugehörigen Testfälle ein falsches Gefühl der Sicherheit bezügliches des geprüften Verhaltens vermittelt. Das Risiko solcher Fehler wird durch den Einsatz von `TDD` verstärkt, da die Entwicklung der Implementierung sich hier wie beschrieben gänzlich nach dem durch die Tests geforderten Verhalten richtet.

Ebenso wie das Schreiben von Tests ist auch deren Verifikation allein durch individuelle Inspektion von simuliertem Waveform-Output nicht praktikabel, sowohl aus Zeitgründen, als auch wegen der erneuten Möglichkeit menschlichen Versagens bei der Auswertung.

Um diese beiden Problemstellungen -also das Erzeugen einer großen Menge fehlerfreier Tests in kurzer Zeitspanne und deren effiziente und korrekte Verifizierung- zu lösen haben wir bereits zu Beginn der Entwicklungsphase Konzepte zur automatischen Testgenerierung und -verifikation entwickelt und diese durch eine Reihe von Skripten umgesetzt, deren Funktionalität im Verlauf des Projektes in Anpassung an neu hinzukommende Instruktionen stetig verbessert und erweitert wurde und die letztendlich die Effizienz des Entwicklungsprozesses drastisch gesteigert haben. Der folgende Abschnitt beschreibt kurz die Implementierung der für alle Tests eingesetzten Testbench, anschließend wird im Detail auf die Implementierung und Funktionalität dieser Skripte eingegangen.

5.2 TOPLEVEL TESTBENCH

Alle Testfälle sind dem beschriebenen `Black-Box-Testing` Prinzip nach für das `top_z80` Modul implementiert. Somit wird mit jedem Test nicht nur das korrekte Verhalten der CPU selbst sondern auch das der Speicher-Kontrolllogik, des internem Programm- und Datenspeicher sowie

der IO-Pins getestet. Zusätzlich sind in der zugehörigen Testbench ein externer Datenspeicher sowie ein "IO-Speicher" an die zu testende Schaltung angeschlossen (siehe Listing 5.1).

Beide Speicher sind dabei durch das Modul `datamem_mock` modelliert, welches nicht in der synthetisierten Schaltung verwendet wird. Dieses bietet ein einfaches Interface zum Schreiben und Auslesen von Daten eines Speichers variabler Größe. Außerdem implementiert es einen einfachen Plausibilitäts-Check für Speicherzugriffe. Dabei werden bei Schreib- und Leseoperationen nur dann valide Daten geschrieben bzw. ausgegeben, wenn die anliegende Speicheradresse mit der vor der letzten Aktivierung des Chip-Enable Eingangs anliegenden Adresse übereinstimmt. Kompliziertere Fehler wie das zu kurze Halten der Speicheradresse **nach** erfolgter Schreiboperation werden an dieser Stelle nicht behandelt.

Durch den "IO-Speicher" werden externe Geräte modelliert. Dass bei Input- und Output Instruktionen korrekte Daten ausgegeben und eingelesen werden, kann mithilfe dieses Speichers einfach überprüft werden indem das `n_iorq` Signal als Chip-Enable für die entsprechende `datamem_mock` Instanz genutzt wird.

Listing 5.1: Externer Datenspeicher und "IO-Speicher" (Ausschnitt `tb_top_z80.v`)

```
datamem_mock #(
    .SZ_LOG2(EXT_DATAMEM_BYTES_LOG2),
    .INITFILE("datamem_ext.txt")
) datamem_ext_i (
    .clk(i_clk),
    // Inputs
    .addr(o_addr[EXT_DATAMEM_BYTES_LOG2-1:0]),
    .din(io_data),
    .ce(~o_n_mreq),
    .we(~o_n_wr),
    // Outputs
    .dout(io_data)
);

datamem_mock #(
    .SZ_LOG2(IOMEM_BYTES_LOG2),
    .INITFILE("iomem.txt")
) iodev_mock_i (
    .clk(i_clk),
    // Inputs
    .addr(o_addr[IOMEM_BYTES_LOG2-1:0]),
    .din(io_data),
    .ce(~o_n_iorq),
    .we(~o_n_wr),
    // Outputs
    .dout(io_data)
);
```

Zusätzlich existieren in der Testbench "Swap-Speicher" für Daten- und IO-Speicher. Diese Swap-Speicher werden benötigt, da während der automatisierten Testfälle schreibende Speicherzugriffe nach jedem Test rückgängig gemacht werden müssen. Andernfalls könnten Abhängigkeiten zwischen den einzelnen Tests eines Testfalls entstehen, wenn in einem Test ein in einem anderen geschriebenes Byte eingelesen wird. Dies sollte möglichst vermieden werden um die Test-Generierung und letztlich auch das Debuggen fehlerhafter Instruktionen durch Analyse der Testergebnisse zu erleichtern. Die Swap-Speicher werden mit den gleichen Inhalten initialisiert wie die tatsächlichen Speicher-Instanzen. Der Daten-Swap-Speicher wird dabei aus den Inhalten des internen und externen Datenspeicher zusammengesetzt. Nach einem schreibenden Speicherzugriff kann dann die beschriebene Stelle im betroffenen Speicher mithilfe des entsprechenden Swap-Speichers zurückgesetzt werden.

Vor Ausführung der Testfälle übernimmt die Testbench zudem die Initialisierung des Programmspeichers mittels des von uns dafür vorgesehenen Mechanismus. Dazu wird der Inhalt des internen Programmspeichers zunächst in ein Hilfs-Speicherelement in der Testbench geladen. Anschließend wird dieser Byte für Byte an den Datenbus-Eingang des Chips angelegt

(nach Deaktivierung des Reset-Signals). Nachdem der interne Programmspeicher eingelesen ist beginnt die CPU automatisch mit der Ausführung des geladenen Programms (siehe Abschnitt 3.2.1). Der relevante Code ist in Listing 5.2 gezeigt.

Listing 5.2: Initalisierung des internen Programmspeichers (Ausschnitt `tb_top_z80.v`)

```
// externally initialize program memory
reg [7:0] progmem [PROGMEM_BYTES-1:0];
reg [PROGMEM_BYTES_LOG2:0] progmem_counter;

initial begin
    $readmemh("progmem.txt", progmem);
    progmem_counter = 0;
end

always @(posedge i_clk) begin
    # 10

    if (progmem_counter < PROGMEM_BYTES) begin
        din_init <= progmem[progmem_counter];
        progmem_counter <= progmem_counter + 1;
    end
    else begin
        din_init <= 8'hzz;
    end
end

assign io_data = din_init;
```

Die Ausgänge von externem Daten- und IO-Speicher sowie dem Initialisierungs-Signal sind alle mit dem Datenbus-Eingang des Chips verbunden, nach Abschluss der Initialisierung geht letzteres in einen High-Impedance Zustand über (ebenso die Speicher wenn der entsprechende Chip-Enable Eingang nicht aktiv ist).

5.3 AUTOMATISCHE TESTGENERIERUNG

Inspiration für die automatisierte Generierung von Tests war das von Frank D. Cringle geschrieben ZEXDOC Programm[1], welches gezielt alle Instruktionen des Z80 abarbeitet, dabei in regelmäßigen Abständen eine Prüfsumme über den als binären String kodierten Prozessorzustand bildet und diese mit einem bei Durchlauf des Programms auf echter Hardware ermittelten Wert vergleicht.

Dieser Ansatz war für unsere Zwecke noch nicht flexibel genug. Zum einen setzten das wiederholte Herstellen eines bestimmten Prozessorzustands und das Berechnen der Prüfsumme bereits die korrekte Funktion einer großen Zahl von Instruktionen voraus, zum anderen lässt sich bei fehlerhafter Prüfsumme nicht sofort feststellen, welche Instruktion den Fehler hervorgerufen hat und welche Untermenge des Prozessorzustands betroffen ist. Das Programm eignet sich also zwar um die Konformität einer bestehenden kompletten Implementierung zu testen, aber in dieser Form nur bedingt zur Fehlerdiagnose und somit auch nicht als Grundlage des TDD Ansatzes.

Wir haben die Probleme umgangen, indem wir das Verhalten des simulierten Verilog-Codes mit dem eines Software Z80 Simulators verglichen haben. Für jede Kombination von System-Ausgangszustand und angewandter Instruktion kann der korrekte resultierende Systemzustand leicht und im Detail mittels des Simulators ermittelt werden und als Grundlage für die Erzeugung eines Tests dienen. Derartige Tests erlauben es Fehler in der Implementierung zum frühestmöglichen Zeitpunkt zu erkennen und genau zu lokalisieren. Die Analyse und Korrektur von Fehlern wird damit erheblich vereinfacht.

[1] Zum Beispiel verfügbar unter: `ftp://ftp.ping.de/pub/misc/emulators`.

Der von uns für diesen Zweck gewählte Simulator ist das Programm z80sim, dessen Quellcode als Teil des von Udo Munk entwickelten Softwarepaketes z80pack offen verfügbar ist[2]. Dieser Simulator eignet sich hier vor allem aufgrund seines intuitiven Kommandozeilen-Interfaces, welches das Anbinden an andere Software-Tools möglich macht. Aufgrund des relativ hohen Bekanntheitsgrades und der jahrzehntelangen Entwicklung von z80pack ist außerdem anzunehmen, dass vom Simulator produzierte Ergebnisse das Verhalten tatsächlich produzierter Z80 Prozessoren weitestgehend korrekt abbilden.

Das von uns entwickelte ictest[3] Skript generiert unter Nutzung des Software-Simulators aus einer kompakten Testfall-Beschreibung in einem von uns spezifizierten Format automatisch einen kompletten Testfall. In jedem von icncsim prepare -tc=... erzeugten Testfall-Verzeichnis ist eine solche Testfall-Beschreibung in Form einer zex.txt Datei angelegt. Jede zex.txt Datei enthält dabei mindestens folgende Elemente (in beliebiger Reihenfolge, jedes auf einer eigenen Zeile in der Form ELEMENTNAME: ELEMENT):

name Ein beliebiger eindeutiger Identifier für den Testfall.

Beispiel: name: alu8i

desc Ein beliebiger beschreibender String der angeben sollte, welche Instruktion(en) getestet werden.

Beispiel: desc: 'ALUOP [A,]n'

base Ein Bitvektor.

Mit dem base-Vektor lassen sich für einen einzelnen Test sowohl die auszuführende Instruktion als auch der Systemzustand vor Ausführung des Tests spezifizieren.

Der Vektor setzt sich dabei aus Elementen der Form BEZEICHNER:WERT zusammen, wobei BEZEICHNER ein Identifikator für ein Instruktions-Byte oder ein CPU-Register und WERT ein binärer oder hexadezimaler Wert ist (die Unterscheidung erfolgt mittels b bzw. h Suffixen). Einzelnen Bits bzw. Nibbles kann hier neben 0-1 bzw. 0-f auch ein x zugewiesen werden, die so markierten Bits werden dann bei der Testfall-Generierung zufällig bestimmt.

In der durch ictest erzeugten testcase.v Datei wird der so beschriebene partielle Systemzustand vor dem ersten zum Test gehörigen Clock-Zyklus durch explizites Setzen der entsprechenden internen Datenpfad-Elemente hergestellt. Zudem wird die beschriebene Instruktion an geigneter Stelle in der Datei progmem.txt hinterlegt.

Aus den im Software-Simulator für die gleiche Kombination von Ausgangs-Systemzustand und auszuführender Instruktion ermittelten resultierenden Werten für die im base-Vektor vorkommenden Register werden automatische Checks in die testcase.v Datei eingefügt, die bei abweichenden Ergebnissen der Verilog-Simulation entsprechende Fehlermeldungen produzieren[4].

Beispiel: base: INST1:c6h,INST2:xxh,A:xxh,F:xx0x0xxxb

INST1 ist das erste Instruktions-Byte, dem hier ein fester Wert zugewiesen wird (in diesem Fall der Opcode der ADD A,n Instruktion). INST2 ist der zugehörige, zufällig bestimmte, Immediate-Wert. Der Inhalt des Akkumulators vor Ausführung des Tests und alle dokumentierten Flags sind ebenfalls zufällig bestimmt. Nach jedem Test werden der Wert des Akkumulators und alle Flags überprüft.

---

[2] Siehe https://www.autometer.de/unix4fun/z80pack/.

[3] Benannt in Anlehnung an die bestehende Toolchain. ictest bietet neben der Testgenerierung noch weitere, später im Bericht erläuterte Funktionalitäten und ist eigentlich eine Sammlung verschiedener Bash, C-Shell, Expect, Perl und Python Skripte, siehe auch die INSTALLATION.md Datei im Root-Verzeichnis des Projektes.

[4] Siehe Abschnitt 5.4.

Zusätzlich können folgende Elemente enthalten sein:

`cycle` Ein Vektor, der die gleichen Elemente wie der `base`-Vektor enthalten muss.

Der `cycle`-Vektor dient dazu, systematisch Systemzustände auf Basis des `base`-Vektors zu enumerieren um die Inklusion zugehöriger Tests im erzeugten Testfall sicherzustellen (im Gegensatz zu `repeat`, s.u.).

Ist ein `cycle`-Vektor angegeben, wird zusätzlich zu dem durch den `base`-Vektor beschriebenen Test für alle möglichen Kombination von Bits an den Stellen im `base`-Vektor, für die das entsprechende Bit im `cycle`-Vektor gesetzt ist, ein zusätzlicher Test erzeugt.

Mit dem `cycle`-Vektor lassen sich so Gruppen von Tests für ähnliche Instruktionen und Instruktionen, in deren Opcodes Register kodiert sind, kompakt beschreiben. Da die Anzahl der Tests, die aus einer `zex.txt` Beschreibung mit angegebenem `cycle`-Vektor erzeugt werden, exponentiell mit der Anzahl im `cycle`-Vektor gesetzten Bits zunimmt, muss allerdings fast immer ein Kompromiss bezüglich der Vollständigkeit der so beschriebenen Testgruppen gemacht werden.

Beispiel: `cycle: INST1:38h,INST2:00h,A:00h,F:00000000b`

Dieser `cycle`-Vektor beschreibt zusammen mit obigem `base`-Vektor zusätzlich zur `ADD A,n` Instruktion sämtliche weiteren 8-Bit ALU Operationen mit Immediate-Operand.

`repeat` Eine positive Ganzzahl.

Der Basisvektor wird bei angebenem `repeat`-Wert zusätzlich `repeat`-mal instanziiert. Dabei werden zufällige Bits jeweils neu bestimmt. So lassen sich z.B. wenn eine explizite Enumerierung von Tests mittels eines `cycle`-Vektors nicht sinnvoll ist, dennoch Gruppen ähnlicher Tests generieren. Dabei ist offensichtlich nicht garantiert, dass die so generierten Tests alle relevanten Grenzfälle abdecken, daher sollte eine ausreichend große Menge von Tests generiert werden.

Dieser Mechanismus kann auch im Zusammenspiel mit einem `cycle`-Vektor verwendet werden, in diesem Fall wird jeder durch den beschriebenen `cycle`-Mechanismus erzeugte Test zusätzlich `repeat`-mal instanziiert.

`memcheck` Eine Liste von zu überprüfenden Speicheradressen.

Der Wert der entsprechenden Stellen im Speicher nach Ausführung jedes generierten Tests wird mit dem vom Software-Simulator ermittelten verglichen. Da ein Vergleich des gesamten Speicherinhaltes -entweder explizit oder über eine Prüfsumme- nach jedem Test aufgrund der Größe des Adressbereiches technisch nicht realisierbar ist, bietet `memcheck` eine Möglichkeit für Instruktionen, die schreibend auf den Speicher zugreifen, den korrekten Inhalt wichtiger Speicheradressen zu verifizieren.

Beispiel: `memcheck: (IX+INST3),(IY+INST3)`

Eine typische `memcheck` Liste die so in fast allen `zex.txt` Dateien, die Testfälle für Instruktionen mit indizierter Adressierung beschreiben, vorkommt. Nach Ausführung jedes Tests wird der Wert an der Speicheradresse, welche durch den Inhalt des `IX` bzw. `IY` Registers und den im dritten Instruktions-Byte enthaltenen Offset bestimmt ist, mit dem durch Simulation ermittelten Wert verglichen. Dies ist vor allem auch für dynamisch (z.B. mittels auf `x` gesetzten Bits) generierte Register-Inhalte bzw. Instruktion-Bytes möglich.

`pccheck` `abs` oder `rel`.

Ist `pccheck` gesetzt, so wird jeweils nach Ausführung eines Tests der Wert des Programm-Counters mit dem im Software-Simulator ermittelten Wert verglichen. Somit kann die Funktionalität (bedingter) Sprünge verifiziert werden.

Dabei können in einem Testfall jeweils nur absolute (`abs`) **oder** relative (`rel`) Sprünge überprüft werden. Dies hängt damit zusammen, dass der Software-Simulator vor der Erzeugung jedes einzelnen Tests neu gestartet wird. Hierdurch muss bei relativen Sprüngen immer ein Korrekturfaktor zum ermittelten Erwartungswert für den PC addiert werden, um dem fortlaufenden PC bei der Simulation des Verilog Testfalls Rechnung zu tragen. Der gleiche Korrekturfaktor muss offensichtlich auch bei absoluten Sprüngen dann genutzt werden, wenn eine bedingte Verzweigung nicht stattfindet. Bei absoluten Sprüngen zur nächsten regulären Instruktion versagt dieser Mechanismus allerdings, da dieser Fall im Software-Simulator nicht von einem nicht ausgeführten Sprung unterschieden werden kann und der Korrekturfaktor folglich fäschlicherweise zum Erwartungswert addiert wird. Tritt dieser Fall bei der Testfall-Generierung zufällig ein (durch in der `zex.txt`-Datei auf `x` gesetzte Bits/Nibbles), muss der gesamte Testfall neu generiert werden.

`flagignore` Eine 8-Bit Maske.

Mit `flagignore` lässt sich bei Angabe des Flag-Registers im `base`-Vektor selektiv die Überprüfung einzelner Flag-Bits deaktivieren. Dies ist notwendig, da einige Flag-Bits nach Ausführung bestimmter Instruktionen einen laut Dokumentation undefinierten Wert annehmen können. D.h. ob und wie diese Flags von den entsprechenden Instruktionen gesetzt werden, ist der konkreten Implementierung überlassen und sollte daher von einem `Black-Box-Test` auch nicht überprüft werden.

Beispiel: `flagignore: 80h`

Mit dieser `flagignore`-Maske wird der Wert des Sign-Bits nicht getestet.[5]

`skip` Eine Liste von Systemzuständen, für die kein Test generiert werden soll.

Mit `skip` können bestimmte Werte für eine Reihe von Elementen des `base`-Vektors angegeben werden, die bei der Generierung von Tests mittels eines `cycle`-Vektors oder `repeat` übersprungen werden müssen. Dabei hat ein auf `x` gesetztes Bit/Nibble in diesem Fall einen *don't care* Effekt. Dies kann z.B. notwendig sein um die Generierung von `HALT` Instruktionen zu verhindern oder `SP > 0x0001` zu garantieren damit der Stack nicht in Adressbereiche vordringt, die im Software-Simulator nicht zur Ablage von Daten nutzbar sind.

Beispiel: `skip: INST1:{01xxx100b;0110xxxb}`

Hiermit werden garantiert keine `base`-Vektor Varianten erzeugt, in denen das erste Byte der Instruktion einen der Werte `0x01000100`, `0x01001100`, `...` oder `0x0110000`, `0x0110001`, `...` annimmt.

`ictest` kann eine solche `zex.txt` Datei einlesen und aus dieser automatisch eine entsprechende `testcase.v` Datei sowie Initialisierungs-Dateien für internen und externen Programm- und Datenspeicher generieren. Die für die `testcase.v` Datei benötigten erwarteten Systemzustands-Übergänge werden bestimmt, indem für jeden im Testfall enthaltenen Test eine Instanz von `z80sim` gestartet wird, die mittels eines `Expect`-Skripts gesteuert zunächst Initialisierungs-Code (um den Simulator in den richtigen Ausgangs-Systemzustand zu bringen) und anschließend die zu testende Instruktion ausführt. Der resultierende Systemzustand wird mittels regulärer Ausdrücke aus dem Simulator-Output geparst.

Auf der Kommandozeile geschieht dies durch Aufruf von `ictest generate TESTCASES`, wobei `TESTCASES` Platzhalter für eine beliebige Anzahl von durch `icncsim prepare -tc=...` erzeugte Testfall-Verzeichnisse mit enthaltenen `zex.txt` Dateien ist.

---

[5] Die zwei reservierten Bits im Flag-Register werden unabhängig von der `flagignore`-Maske in jedem Fall nicht getestet.

Bei Erzeugung mehrerer Testfälle kann dieser Schritt merkbar langsam sein. Es wäre hier eventuell vorteilhaft gewesen, die Interaktion mit dem Simulator direkt über Zugriff auf dessen C-Interface zu realisieren. Da jeder Testfall aber nur bei Änderungen an der `zex.txt` Datei oder signifikanter interner Umstrukturierung der Verilog-Implementierung (z.B. Umbenennen von Registern) neu generiert werden muss, ist Performance hier nicht kritisch. Die Anzahl der erzeugten Tests pro Testfall sollte außerdem 1000 nicht stark überschreiten, da ansonsten unter Umständen die Makefile Erzeugung mittels `icncsim` fehlschlagen kann.

Einige wenige Tests können nicht automatisch generiert werden. In diesen Fällen enthält die `zex.txt` Datei nur das Stichwort `MANUAL` und wird von `ictest` ignoriert. Die `testcase.v` Datei und die Speicher-Inhalte müssen dann jeweils in einer mit der automatischen Verifizierung (siehe nächster Abschnitt) kompatiblen Form von Hand geschrieben werden. Dies betrifft zum Beispiel Interrupts und IO mit externen Geräten, beides wird nicht direkt vom Software-Simulator unterstützt.

Die Testfälle können mittels `ictest run TESTCASES` ausgeführt werden. Bei Bestehen eines Testfalls sollte eine Konsolen-Ausgabe ähnlich der folgenden entstehen:

```
=== Executing testcase top_z80_alu8i_tc ===
Generating makefile
Running simulation
Running testcase 'alu8i ('ALUOP [A,]n')'
Testcase 'alu8i ('ALUOP [A,]n')' done
==> no failures
```

Schlägt einer der Tests innerhalb des Testfalls fehl, wird der Testfall an dieser Stelle abgebrochen und auf der Konsole wird z.B. Folgendes ausgegeben:

```
=== Executing testcase top_z80_alu8i_tc ===
Generating makefile
Running simulation
Running testcase 'alu8i ('ALUOP [A,]n')'
Assertion failed: A (Testcase #0 0d4c688f, expected 0xbb but got 0xba)
==> 1 failure(s)
```

Dieser Konsolenausgabe ist zu entnehmen, welcher Testfall den Fehler hervorgerufen hat (hier der erste) und worin genau dieser besteht. In diesem Fall weicht der Wert des Akkumulator-Registers vom erwarteten Wert (`0xbb`) ab. Die Zahlenfolge hinter der Testcase-Nummer ist ein eindeutiger Identifier (hier verkürzt) und dient dazu, den zur Meldung gehörenden Check innerhalb des fehlschlagenden Tests schnell in der `testcase.v` Dateie wiederfinden zu können.

Der folgende Abschnitt erläutert die bei Ausführung von `ictest run` im Hintergrund ablaufenden Prozesse.

5.4 AUTOMATISCHE TESTVERIFIZIERUNG

Eine von `ictest generate` aus einer `zex.txt` Beschreibung generierte `testcase.v` Datei könnte zum Beispiel folgende Zeile enthalten:

```
unit.assert_eq8(8'hBA, top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.a,
    "A (Testcase #0 0d4c688f-8c88-40ba-a980-7ebf75f5a7ee)");
```

Hier kommt das für diesen Zweck geschrieben `vunit`-Modul zum Einsatz (hier als `unit` instanziiert), welches eine Reihe von Tasks definiert, die jeweils eine Bedingung überprüfen (in den meisten Fällen die Gleichheit zweier ein-, acht- oder 16-Bit Werte) und bei Nichterfüllung dieser Bedingung eine diagnostische Nachricht auf dem Terminal ausgeben. Der Test wird im Falle der

`assert*` Tasks an dieser Stelle abgebrochen. Mit den `expect*` Tasks können potentiell auch mehrere Fehler in Folge diagnostiziert werden, diese finden in der letztendlichen `ictest` Implementierung allerdings keine Verwendung. Bei Ausführung von `ictest run` wird der Test im Batch-Modus ausgeführt und der entstehende Output von einem Perl Script geparst, das die so entstandenen Fehlermeldungen aus dem gesamten entstehenden Output herausfiltert und aufbereitet.

Listing 5.3 zeigt einen vollständigen Test. Zunächst werden in einige Register entsprechend des diesem Test zugrunde liegenden Systemzustands-Vektor auf ihre Ausgangswerte gesetzt. Anschließend schreitet die Simulation in solange Takt für Takt voran, bis der Instruction Fetch Zustand der Instruktion direkt nach der zu testenden erreicht ist. Dabei wird ein Zähler eingesetzt, um die Simulation notfalls zu beenden, falls die getestete Instruktion durch einen Fehler in der FSM nicht in einer angemessener Anzahl von Takten terminiert. Danach folgen Assertions für alle zuvor gesetzten Register und die Flag-Bits. Zuletzt wird der an einer Adresse im externen Datenspeicher liegende Wert überprüft und danach mithilfe des Swap-Speichers auf seinen Zustand vor Ausführung des Tests zurückgesetzt.

Listing 5.3: Vollständiger Test (Ausschnitt `top_z80_rotrhl_tc/testcase.v`, leicht angepasst)

```
// BEGIN TESTCASE #0
top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.a = 8'hF1;
top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.f = 8'h42;
{ top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.b, top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.c } = 16'h7299;
{ top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.d,  top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.e } = 16'hF25E;
{ top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.h, top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.l } = 16'h07AB;

#(CLKPERIOD);

loop_condition = 1;
loop_counter = 0;
while (loop_condition) begin
    #(3 * CLKPERIOD / 4);
    if (top_z80_i.z80_i.cpu_i.controller_i.fsm_i.state == FSM_STATE_INSTR_FETCH1_1) begin
        loop_condition = 0;
    end else begin
        #(CLKPERIOD / 4);
        loop_counter = loop_counter + 1;
        if (loop_counter == 32)
            unit.fail("Instruction not terminating (Testcase #0)");
    end
end

unit.assert_eq8(8'hF1, top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.a,
"A (Testcase #0 291bbb07-bcc9-47d9-b4b7-09d062abd07a)");
unit.assert_eq16(16'hE499, { top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.b,
                             top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.c },
"BC (Testcase #0 9b91f096-523c-4f9e-9810-5ee1b656838c)");
unit.assert_eq16(16'hF25E, { top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.d,
                             top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.e },

// ...

"S (Testcase #0 7981d155-ee97-4b19-bdf9-241d4ee32d73)");
unit.assert_eq(1'b0, top_z80_i.z80_i.cpu_i.datapath_i.regfile_i.f[6],
"Z (Testcase #0 e169f6f1-93a7-4e25-919c-940a357e7c01)");

// ...

unit.assert_eq8(8'h15, datamem_ext_i.datamem[13'h7AB],
"EXT DMEM (Testcase #0 7d481749-8d95-4f41-a6bb-6d8cbf38497b)");
datamem_ext_i.datamem[13'h7AB] = datamem_swap[16'h7AB];

// ...
```

5.5 TEST-CODE-COVERAGE ANALYSE

Mit `ictest` können ebenfalls automatisiert die Coverage-Daten für eine beliebige Untermenge aller Testfälle zusammengeführt werden. Dazu muss `ictest coverage TESTCASES` aufgerufen werden, dabei wird automatisch `iccr` gestartet. So kann nach Ausführen aller Testfälle mittels `ictest coverage top_z80*` die komplette erzeugte Code-Coverage visualisiert werden. Die von uns entwickelten Tests erreichen dabei praktisch vollständige Abdeckung. Ausnahme sind eine Reihe von impliziten `case` Statement `defaults` und `else` Zweigen -die für die Funktionalität der Implementierung nicht relevant sind- sowie einige wenige nicht aufgetretene Signalflanken. Die hohe Code-Coverage ist in Abbildung 5.1 belegt.

Abbildung 5.1: Test-Coverage Übersicht

Diese allein ist kein Garant dafür, dass die funktionale Verifikation vollständig ist. In Kombination mit der sehr großen Zahl automatisch generierter Tests kann aber mit entsprechender Sicherheit davon ausgegangen werden, dass die Implementierung das in der Dokumentation spezifizierte Verhalten in hohem Maß erfüllt.

5.6 VOLLSTÄNDIGE TESTPROGRAMME

Die betrachteten Methoden der Verifizierung basieren auf der Annahme, dass es keine fehlerhaften Instruktionen gibt, die Teile des Systemzustandes verändern, auf die sie eigentlich keinen Einfluss haben sollten (z.B. schreibende Speicherzugriffe bei Instruktionen, die nur lesend auf den Speicher zugreifen sollten). Derartige Fehler können durch die automatisierten Tests nicht im Allgemeinen entdeckt werden. Dies wäre auch schwer umsetzbar, da das Überprüfen des gesamten Systemzustandes die Laufzeit der Testfälle sprengen würde.

Obwohl Fehler dieser Art unwahrscheinlich sind, ebenso wie die Möglichkeit verbleibender fehlerhafter Abhängigkeiten zwischen Instruktionen[6], ist es daher sinnvoll die Implementierung zusätzlich zu testen, indem mit dieser einige komplexere Programme ausgeführt werden.

Dies ist außerdem die effektivste Methode um sicherzustellen, dass die Ergebnisse des Synthese- bzw. Layout-Schrittes ebenfalls fehlerfrei sind. Aufgrund der Transformation der abstrakten Beschreibung des Hardwareverhaltens in konkrete Gatter-Instanzen und die automatische Umbenennung von internen Signalen müssten entsprechend angepasste Varianten aller automatisierten Tests erzeugt werden. Wesentlich einfacher ist es, nur das Verhalten oben genannter Testprogramme zu analysieren. Produzieren Synthese bzw. Layout Ergebnis bei Ausführung dieser Programme die erwarteten Ergebnisse, kann davon ausgegangen werden, dass diese ebenfalls funktional korrekt sind.

Um die gewünschte Komplexität der genutzten Testprogramme zu erreichen ohne übermäßig Zeit in deren Entwicklung zu investieren wurden diese mithilfe des *Small Device C Compilers*[7]

[6] Da die Tests jeweils komplette Systemzustands-Übergänge verifizieren.

[7] Kurz SDCC, siehe `http://sdcc.sourceforge.net/`.

aus High-Level C code erzeugt.

Das Haupt-Testprogramm führt die LUP Matrix-Dekomposition aus, bildet anschließend eine Prüfsumme über die modifizierten Matrixelemente und legt diese mittels inline Assembly im 16-Bit Register `BC` ab. Das Programm ist ein Überbleibsel aus der Lehrveranstaltung `System- und Schaltkreisentwurf` und musste leicht modifiziert werden, damit daraus für unsere Zwecke nutzbaren Z80 Bytecode erzeugt werden kann[8].

Alle genannten Programme befinden sich im Testfall-Verzeichnis des `top_z80` Moduls in den Verzeichnissen `testprog_...`. Mit `ictest run testprog_...` bzw. `ictest run lay testprog_...` kann überprüft werden, ob das Synthese- bzw. Layout Ergebnis die Programme korrekt ausführen.

[8] Unter anderem durch Elimination dynamischer Speicherverwaltung sowie manueller Implementierung von Multiplikations- und Divisions-Operationen.

6 SYNTHESE UND OPTIMIERUNG

Mit der initialen Implementierung[1] konnte in der Synthese eine Taktrate von ungefähr 135MHz erreicht werden. Ohne Einführung neuer Zustände oder zusätzlicher Datenpfad-Elemente ließ sich diese bis auf knapp über 150MHz optimieren. Dazu wurden im Wesentlichen die Zustände und Opcode-Präfixe sowie die Auswahl-Eingänge aller Multiplexer auf dem kritischen Pfad 1-aus-n kodiert um die kombinatorische Vorzögerung durch den Decoder und die entsprechenden Multiplexer zu verringern.

Diese Methode stieß jedoch schnell an ihre Grenzen; Abbildung 6.1 zeigt schematisch den aus Timing-Reports ermittelten und auch nach wiederholter Synthese (nach Anwendung genannter Optimierungen) persistenen kritischen Pfad durch die initiale Implementierung.

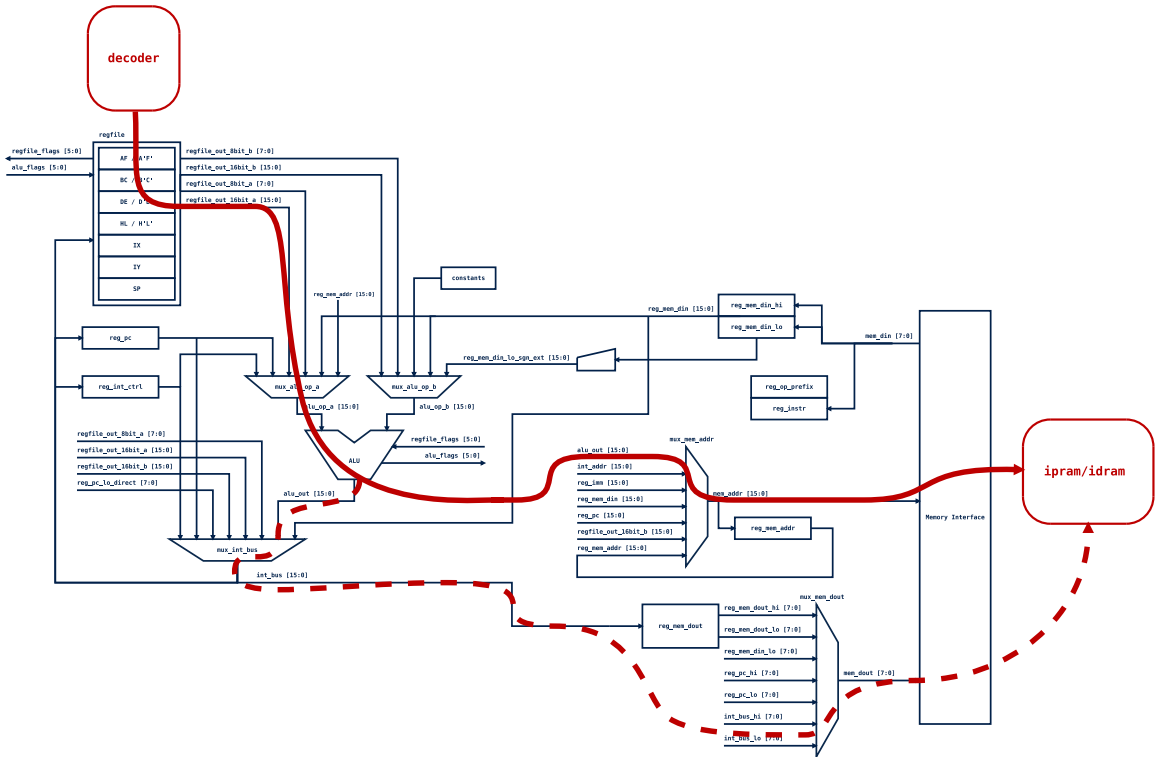

Abbildung 6.1: Initialer Kritischer Pfad

Dieser führt vom Decoder über die ALU und den Speicheradressen-Multiplexer zu einem der internen Speicher[2]. Dieser Pfad wird unter anderem von Instruktionen, welche indizierte Adressierung nutzen, benötigt. Während dieser wird der eingelesene Offset im gleichen Takt, in dem er verfügbar wird, zu einem der Register `IX` bzw. `IY` addiert und das Ergebnis direkt an die Speicherverwaltung übergeben. Diese entscheidet dann, ob der Chip-Enable Eingang des internen Datenspeichers aktiviert werden muss. Bei relativen Sprüngen wird der analoge Pfad zum Programmspeicher genutzt.

[1] Mit minimaler Takt-Länge (fast) aller Instruktionen, siehe Abschnitt 2.

[2] Teils der interne Programmspeicher, teils auch der interne Datenspeicher, je nach Synthese-Lauf.

Der gestrichelte Pfad führt ebenfalls über die ALU zum internen Datenspeicher. Über diesen kann das obere oder untere Byte eines ALU-Ergebnisses im gleichen Takt, in dem es berechnet wird, an den Datenausgang `mem_dout` angelegt werden. In den Synthese-Läufen war der für diesen Pfad ermittelte Slack stets sehr nah an dem des kritischen Pfades. Dies legt nahe, dass auch dieser Pfad einer wesentlichen Erhöhung der Taktfrequenz im Wege steht.

Um diese beiden Pfade aufzubrechen wurden die in Abbildung 6.2 gezeigten Änderungen am Datenpfad vorgenommen.

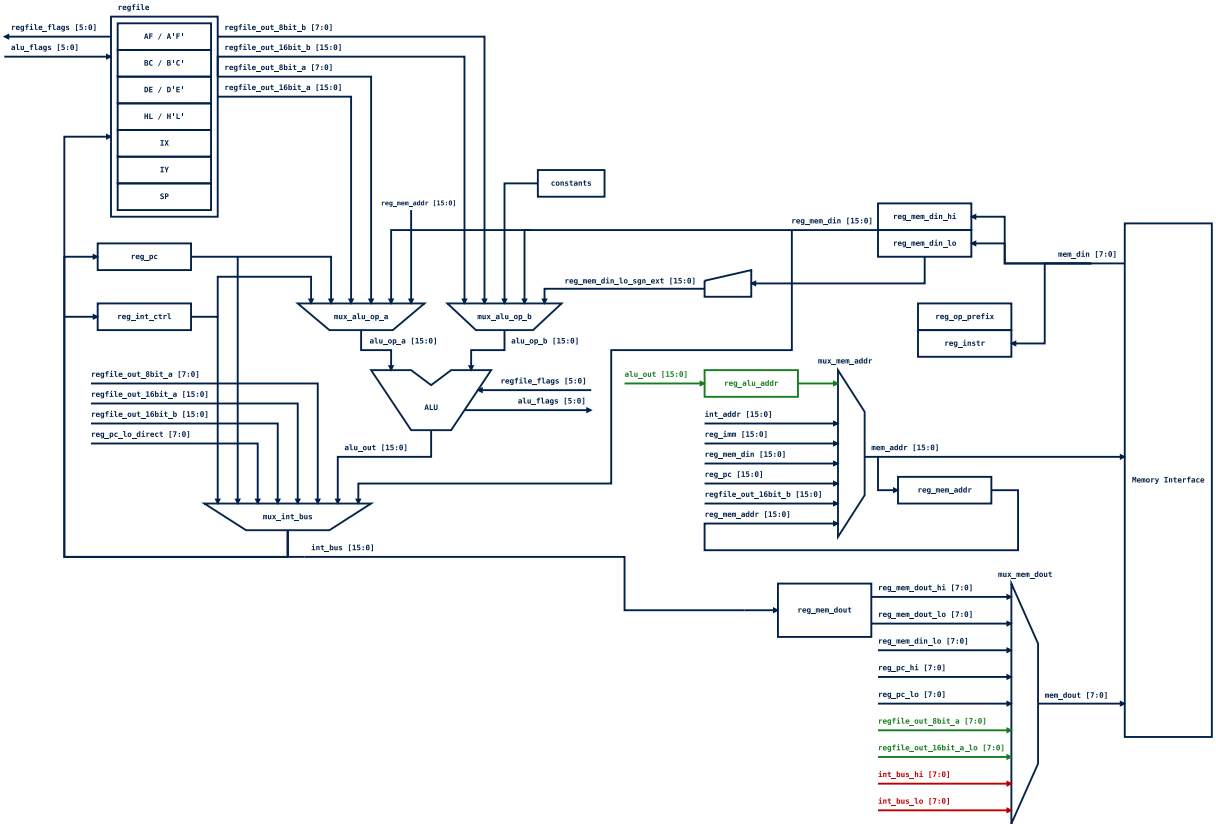

Abbildung 6.2: Optimierter Datenpfad

Dabei wurde das zusätzliches Register `reg_alu_addr` zwischen dem Ausgang der ALU und dem Speicheradressen-Multiplexer eingefügt und der direkte Pfad über die ALU zum `mem_dout` Bus entfernt. Der interne Datenbus liegt jetzt nur noch über das `reg_mem_dout` Register am `mux_mem_dout` Multiplexer an, dafür wurden neue direkte Verbindungen vom Register-File zu diesem hinzugefügt, um Register-Inhalte weiterhin unverzögert auf den `mem_dout` Bus ausgeben zu können. Da berechnete Adressen und zu schreibende Bytes mit diesen Änderungen nicht mehr ohne Verzögerung am `mem_addr` bzw. `mem_dout` Bus angelegt werden können, musste für von dieser Änderung betroffene Instruktionen jeweils ein "Puffer-Takt" zwischen die die ALU-Operation ausführenden Takte und die folgenden, auf Speicher oder externe Geräte zugreifenden, Maschinen-Zyklen eingefügt werden. Diese Lösung wurde bewusst mit dem Ziel gewählt, die notwendigen Änderungen am bestehenden Verilog-Code so gering wie möglich zu halten, nur so konnten die nötigen Anpassungen an Decoder und FSM zeitlich bewältigt werden.

Die den Puffer-Takten entsprechenden Zustände sind in dem in Abbildung 6.3 dargestellten aktualisierten Zustandübergangsgraphen gezeigt, der Übersichtlichkeit halber sind nur neu hinzukommende Zustandsübergänge von und zu den neuen Zuständen gezeigt.

Nach erneuter Synthese konnte die Taktrate so bis auf rund 165MHz erhöht werden. Der neue kritische Pfad ist als Auszug aus dem Timing-Report der Synthese in Listing 6.1 und grafisch in

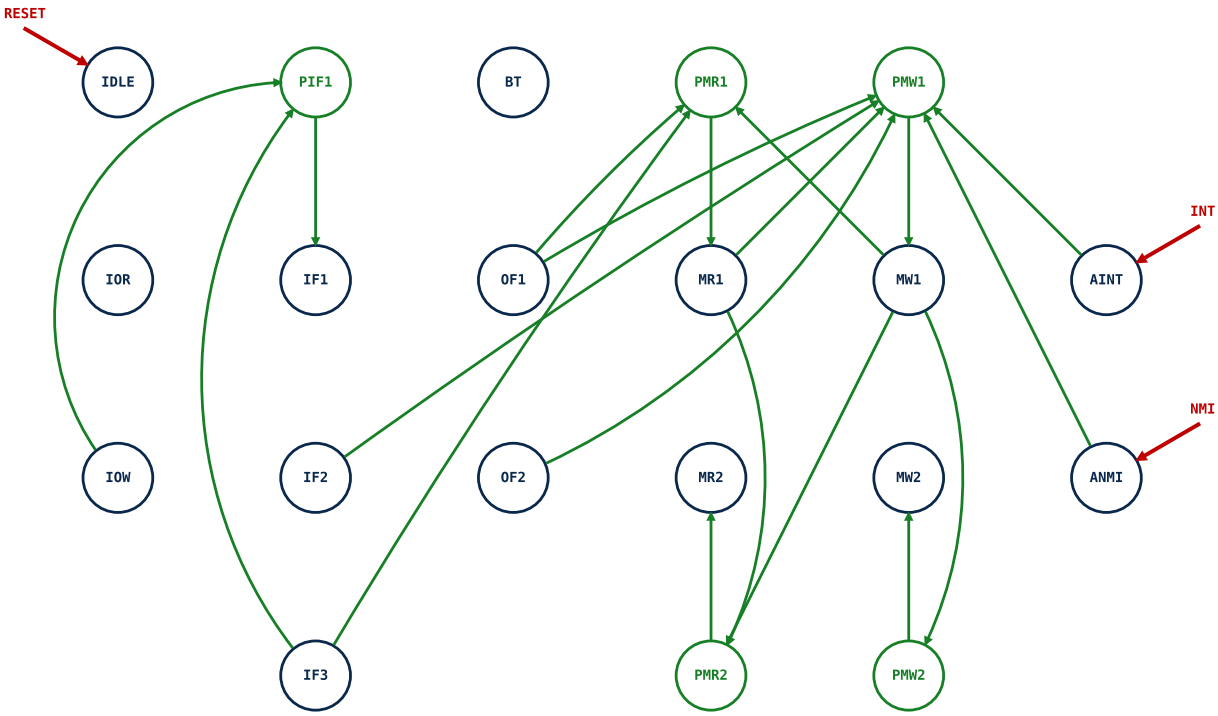

Abbildung 6.3: Zustandsübergänge der Puffer-Zustände

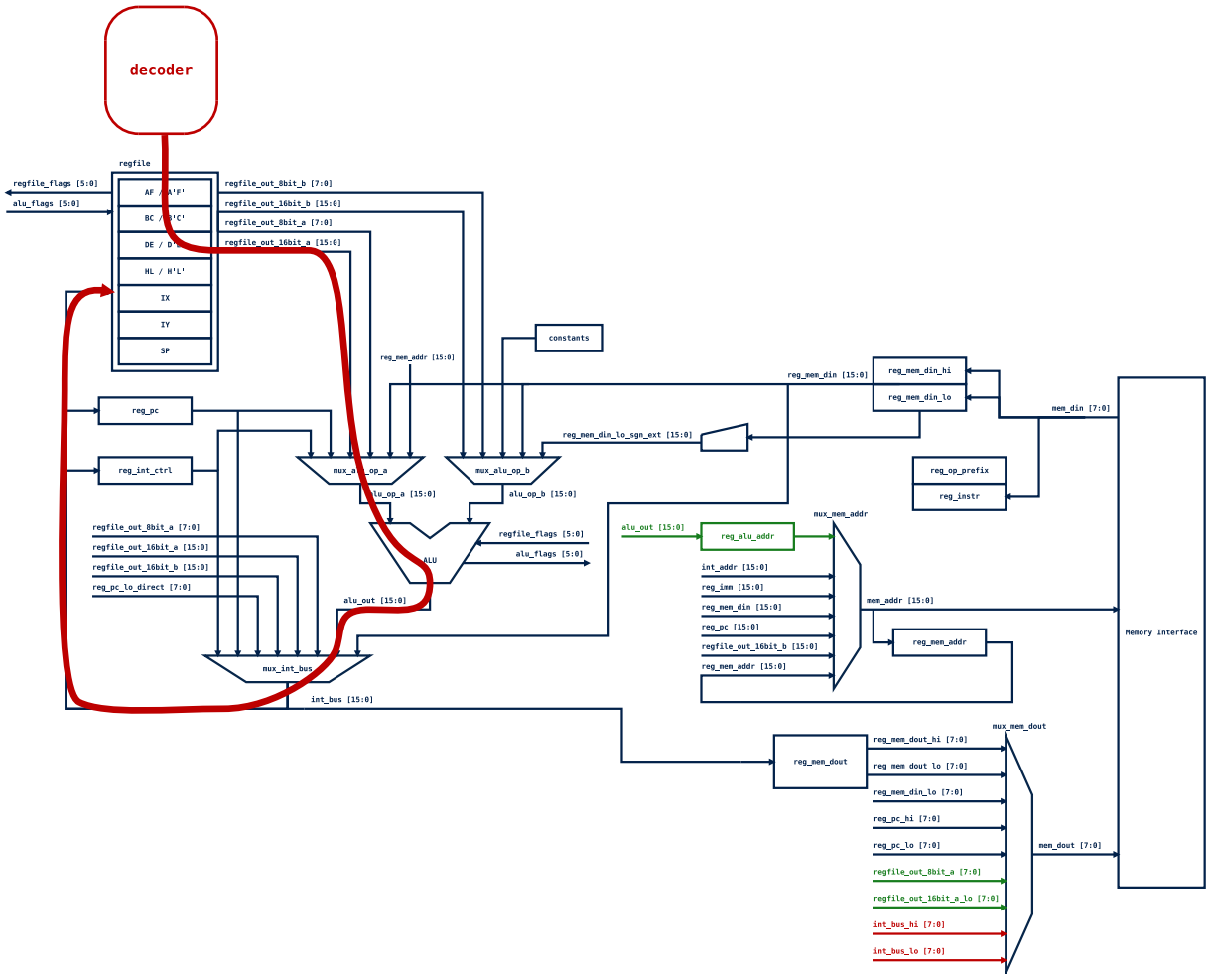

Abbildung 6.4: Neuer Kritischer Pfad

Abbildung 6.4 gezeigt und verläuft über die ALU zum Schreib-Eingang des Register-Files. Ein weiteres Aufbrechen dieses Pfades ist nicht praktikabel, da das Einfügen von Registern direkt vor oder hinter der ALU starke Eingriffe in fast allen implementierten Instruktionen notwendig machen und die durchschnittliche Anzahl von Takten pro Instruktion deutlich erhöhen würde. Eine weitere Steigerung der Taktfrequenz wäre somit nur unter erheblicher Neustrukturierung des gesamten Designs denkbar, z.B. durch Ersetzen der 16-Bit ALU durch eine 8-Bit ALU oder die Einführung einer Pipeline. Dies war im Rahmen dieses Beleges nicht mehr umsetzbar.

Listing 6.1: Optimierter kritischer Pfad (Auszug `top_z80.report_timing`)

```
Point                                                                    Incr       Path
---------------------------------------------------------------------------------------
clock CLK (rise edge)                                                    0.00       0.00
clock network delay (ideal)                                              1.50       1.50
...
z80_i/cpu_i/datapath_i/reg_instr[7] (datapath)                           0.00       2.04 f
z80_i/cpu_i/controller_i/reg_instr[7] (controller)                       0.00       2.04 f
z80_i/cpu_i/controller_i/decoder_i/reg_instr[7] (decoder)
...
z80_i/cpu_i/controller_i/decoder_i/alu_mode[10] (decoder)
                                                                         0.00       3.77 f
z80_i/cpu_i/controller_i/alu_mode[10] (controller)                       0.00       3.77 f
z80_i/cpu_i/datapath_i/alu_mode[10] (datapath)                           0.00       3.77 f
z80_i/cpu_i/datapath_i/alu_i/mode[10] (alu)                              0.00       3.77 f
...
z80_i/cpu_i/datapath_i/alu_i/data_out[9] (alu)                           0.00       6.63 r
z80_i/cpu_i/datapath_i/mux_int_bus_i/alu_out[9] (mux_int_bus)
                                                                         0.00       6.63 r
...
z80_i/cpu_i/datapath_i/mux_int_bus_i/data_out[9] (mux_int_bus)
                                                                         0.00       6.91 r
z80_i/cpu_i/datapath_i/regfile_i/reg_in[9] (regfile)
...
data arrival time                                                                   7.19

clock CLK (rise edge)                                                    6.05       6.05
clock network delay (ideal)                                              1.50       7.55
clock uncertainty                                                       -0.20       7.35
...
library setup time                                                      -0.16       7.19
data required time                                                                  7.19
---------------------------------------------------------------------------------------
data required time                                                                  7.19
data arrival time                                                                  -7.19
---------------------------------------------------------------------------------------
slack (MET)                                                                         0.00
```

Ob die erreichte Steigerung der Taktfrequenz von rund 10% das Einfügen der Puffer-Takte wert ist, ist nicht eindeutig zu beantworten. Die Ausführungszeit der meisten betroffenen Instruktionen wird hierdurch verlängert, andererseits ist die Struktur vieler Instruktionen überhaupt nicht von der Änderung betroffen, diese profitieren daher von der höheren Taktfrequenz. Intuitiv überwiegt dieser Vorteil, da beispielsweise arithmetische und logische Operationen mit indizierter Adressierung in einer typischen Anwendung mit deutlich geringerer Häufigkeit auftreten als solche, die nicht auf den Speicher zugreifen oder diesen direkt adressieren. Die erläuterten Optimierungen wurden daher in die finale Implementierung übernommen.

7 PLACE & ROUTE

Abbildung 7.1 zeigt das Ergebnis des Place & Route Schrittes. Der Chip hat eine Gesamtfläche von $1750x1012\mu m^2$ und eine Gatter-Platzierungsdichte von 66.117%. Die Platzierung der Speicherelemente wurde mit Hinblick darauf gewählt, einfaches Routing von den Anschlüssen ins Innere des Chips und zu den IO-Pins zu ermöglichen.

Abbildung 7.1: Place & Route Ergebnis

Die in Listing 7.1 gezeigte Post-Route Zusammenfassung zeigt, dass durch den Place & Route Schritt keine Timing-Verletzungen entstanden sind.

Listing 7.1: Post-Route Ergebnis (Ausschnitt)

```
------------------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------------------

+--------------------+---------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg |reg2cgate| default |
+--------------------+---------+---------+---------+---------+
|           WNS (ns):|  0.330  |  0.425  |  0.743  |  0.330  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |
|          All Paths:|   893   |   429   |   24    |   466   |
+--------------------+---------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 66.117%
Total number of glitch violations: 0
------------------------------------------------------------------------
```

8 ABSCHLIESSENDE BEWERTUNG

Insgesamt ist das erzielte Ergebnis absolut zufriendstellend, als Haupterfolg sehen wir die ausführliche Verifizierung, die uns großes Vertrauen in die Korrektheit unserer Implementierung gibt. Wenn man das Verhältnis der für die vereinzelten manuell und die automatisiert erzeugten Testfälle benötigten Enwicklungsdauer betrachtet, ist zudem davon auszugehen, dass `ictest` trotz anfangs hohem Entwicklungswand insgesamt zu einer enormen Zeitersparnis geführt hat, ohne die im gegebenen Zeitrahmen eine angemessene Verifizierung unmöglich gewesen wäre.

Bei der eigentlichen Implementierung ist der von uns anfangs angedachte Plan zur Verfolgung unseres Design-Ziels aufgegangen. Es konnte eine angemessen hohe Taktfrequenz erzielt werden, bei gleichzeitiger (im Schnitt) niedriger Anzahl von Takten pro Instruktion. Es wäre jedoch unter Umständen sinnvoll gewesen, die Optimierung nicht erst am Ende des Entwicklungsprozesses vorzunehmen, sondern gleich zu Beginn mit verschiedenen möglichen Datenpfaden und deren kritischen Pfaden in Isolation zu experimentieren. Mit dem von uns verfolgten Ansatz konnten während der Optimierung nur noch moderate Änderungen am Datenpfad vorgenommen werden. Ob fundamental verschiedene Datenpfad-Layouts zu besseren Ergebnissen geführt hätten bleibt unklar. Es war uns jedoch zu Anfang noch nicht bewusst, ob dieser zusätzliche Aufwand überhaupt gerechtfertigt gewesen wäre. Letztlich hätten auch vom Datenpfad-Layout unabhängige Faktoren die Taktfrequenz limitieren können.

Im Allgemeinen sind Teile des Designs “organisch” gewachsen. Es wurden zwar vor Beginn der Implementierung Schaltpläne für den Datenpfad und grobe Ablaufpläne für die wichtigsten Instruktionen (d.h. notwendige Zustandsübergänge) angelegt, diese wurden aber immer wieder angepasst und erweitert um neuen, durch komplexere Instruktionen bedingten Anforderungen gerecht zu werden. Es wäre auch denkbar gewesen, mittels einer kompletten Analyse aller Instruktionen und spezieller Prozessor-Funktionalitäten das gesamte Design auszuplanen und erst dann mit der Implementierung zu beginnen (die dann im Idealfall nur noch Formsache gewesen wäre). Wir haben uns aufgrund unserer mangelnden Erfahrung mit dem Entwurf von CISC Prozessoren gegen diesen Ansatz entschieden. Mit hoher Wahrscheinlichkeit hätten ohnehin erst während der Implementierung aufgedeckte Probleme im initialen Design behoben werden müssen.

ABBILDUNGSVERZEICHNIS

LISTINGVERZEICHNIS