Design digitaler Schaltungen

Vorlesung 1

Folie 2

***

Themen des Courses:

Design von digitalen Grundkomponenten

Gatter, Register, Speicherelemente, RAM

Übung: Digital Chipdesign – Design von digitalen Schaltungen auf einem ASIC

Übung einmal in zwei Wochen, zwei Gruppen – Anfang im Mai

Folie 3

***

Einführung

Wir leben in einer analogen Welt - Alle 'Messgrößen' unserer Umwelt (Licht, Töne, Temperatur, elektrische Spannungen, Druck etc..) sind analog.

Die analoge Verarbeitung ist daher natürlich – Signale werden in Spannungen umgewandelt, verstärkt, gefiltert.

Digitale Elektronik ist heute allgegenwärtig – Kommunikation, Computer, Internet. Digitale Elektronik ersetzt Papier, Bleistift, ermöglicht uns die Information zu bearbeiten und zu speichern.

Folie 4

***

Wie verwenden die Zahlen im dezimalen Format und Alphabet. In der digitalen Elektronik benutzen wir die binären Codes.

Es gibt einige Binäre-Formate:

Betrachten wir z.B. acht Bit Zahlen.

Sie bestehen aus acht Stellen die entweder 0 oder 1 sein können. Jede Stelle nennen wir ein Bit.

Es kann hergeleitet werden dass es 256 acht-bit Zahlen gibt. Solch ein Code ist also geeignet um die ganze Zahlen („integers“) von 0 bis 255 zu kodieren.

Es ist nützlich zu wissen:

2^3 = 8

2^4 = 16

…

2^10 =1024

2^(1N) ~ 2^N x 1000

Folie 5

***

Warum werden in digitaler Elektronik binäre Codes benutzt?

Nehmen wir an wir möchten zwei Zahlen bis zur Größe N kodieren.

Es ist leicht zu rechnen, dass wir dafür lnN/lnX stellige Codes brauchen, wo X die Zahl von Ziffern im Code ist. In Fall von Binären Codes habe wir X = 2, also wir haben nur zwei Ziffern 0 oder 1.

Für eine Zahl bis 256 brauchen wir einen achtstelligen Code für X = 2 und etwa fünfstelligen für X = 3.

Wenn wir nur die Zahl von Speicherplätzen betrachten, wäre ein ternärer Code (X=3) besser.

Lass uns jetzt abschätzen wie komplex die Elektronik wäre, die eine Operation zwischen den zwei Zahlen macht.

Wir würden wahrscheinlich die Operation Bitweise durchführen. Die Komplexität eines Bit-Operators könnte im Zusammenhang mit der Größe der Ergebnistabelle stehen. In Fall von Binären Codes haben wir für zwei Eingangsvariablen vier Möglichkeiten, also vier Zeilen. Im allgeneinen Fall haben wir in der Tabelle X^2 Zeilen. Die Elektronik hätte dann die Größe:

Zahl von Stellen im Code * Komplexität der Elektronik

Oder

lnN/lnX * X^2

Diese Funktion hat ein Minimum in der Nähe von Zahl 2. Also der binäre Code ist am effizientesten.

Folie 6

***

Generell werden in digitalen Schaltungen die Ziffern 0 oder 1 in Form von Potentialen dargestellt. Null entspricht eine niedrigen und Eins einem höheren Potential.

Folie 7

***

Betrachten wir wieder die acht-Bit Zahlen.

Die folgenden Operationen zwischen den Zahlen werden oft gebraucht:

Addition, Subtraktion, Vergleich (Größer als, Gleichheit), Multiplikation

Es stellt sich die Frage, wie werden diese Operationen realisiert.

Sehr generell könnte man wie folgend argumentieren.

Das Ergebnis der Addition und Subtraktion sind 8-bit Zahlen (wir vernachlässigen den Übertrag), das Ergebnis der Multiplikation ist 16 Bit Zahl, und die Ergebnisse von Vergleichen sind binäre Zahlen, bzw. eine Boolesche Variablen (wahr nicht wahr).

Betrachten wir den letzten Fall - Komparator.

Folie 8

***

Um den Komparator zu beschreiben, könnten wir eine große Ergebnisstabelle (Wahrheitstabelle) zusammenschreiben wo wir eine Zeile für jede Zahlenkombination A und B haben. Es gibt 256 x 256 = 2^(16) solchen Kombinationen also 2^16 Zeilen.

Folie 9

***

Normalformen

Wie definieren UND (Konjunktion) Funktion (Verknüpfung) von n Variablen als Funktion mit dem Wert 1 wenn alle Variablen 1 sind.

Das Zeichen für Konjunktion ist ^ oder * oder &

Konjunktion entspricht der Umgangssprache: Ergebnis ist wahr (=1) wenn X1 und X2 und … Xn wahr sind.

Wir definieren auch ODER Verknüpfung (Disjunktion) mit dem Ergebnis null nur wenn alle Variablen null sind.

Das Zeichen für Disjunktion ist v oder + oder |

Es entspricht dem Satz: Ergebnis ist wahr (=1) wenn X1 oder … Xn wahr sind.

Folie 10

***

Die Tabelle für Vergleich zwei 8-Bit zahlen können wir wie folgend als Disjunktive Normalform darstellen:

Wir suchen alle Zeilen mit dem Ergebnis 1 – es gibt sie 256. Für diese Zeilen bilden wir eine UND Verknüpfung, die nur für die Variablen-Werte aus der Zeile 1 ergibt:

ZB 0000_1111 0000_1111

Ki = !A7 & !A6 & !A5 & !A4 & A3 & A2 & A2 & A0 & !B0 …

Zeichen ! bedeutet Negation – wir verwenden es überall dort wo die Variable 0 ist. Die Gesamttabelle ist dann ODER Verknüpfung von allen Ki Funktionen.

F = K1 | … K256

Alternative Zeichen für die Negation sind ~, -| oder Oberstrich.

Folie 11

***

Die Normalform kann oft wie folgend vereinfacht werden

Wenn wir z.B. zwei folgende Terme haben

(X & Ai) | (X & !Ai)

sie können wie folgend vereinfacht werden:

X & (Ai | !Ai) = X & 1 = X

Im Beweis haben wir das Distributivgesetz benutzt sowie die Äquivalenz Ai |!Ai = 1 und X & 1 = X

Also, um die normale Form zu vereinfachen, suchen wir die Paare von Termen in der Form (X & Ai) und (X & !Ai). Ai Variable kann in dem Fall weggelassen werden.

Ein Spezialfall dieser Regel ist

(X & Ai) | X = X

(X und „etwas Spezielles“) oder X ist wahr wenn X wahr ist, oder unwahr wenn X unwahr ist.

Wir nennen es Absorptionsregeln.

Wenn die Minimierung nicht mehr möglich ist, haben wir die Minimale Form.

Folie 12

***

Eine Disjunktive Normalform kann Schaltungstechnisch realisiert werden.

Wie Brauchen Logische Elemente (Gatter) die UND, ODER und Negation Erzeigen. Die logischen Niveaus werden fast immer als Potentiale realisiert.

Eine einfache Möglichkeit logische Elemente zu realisieren sind die spannungsgesteuerten Schalter

Ein Schalter ist geschlossen wenn sein Eingangspotential hoch ist, also dem logischen Eins entspricht.

Folie 13 und 14

***

Wie realisieren wir z.B. die UND Funktion von zwei Variablen A, B, C.

Jede Variable ist ein Draht – Kabel („wire“), sein Potential ist der Wert 0 oder 1.

Wir schalten einfach zwei Schalter in Serie, sie werden an die Masse angeschlossen. An der anderen Seite der Schalter haben wir den Ausgang. Zwischen dem Ausgang und einer positiven Versorgungsspan schließen wir einen Widerstand. Wir definieren das Potential um VDD als logische 1 und das Potential um GND als 0.

Nur wenn alle Eingänge Eins sind ist auch der Ausgang null, sonst ist es 1.

So bekommen wir eine Negation von UND Verknüpfung – englisch NAND.

Die Annahme ist hier dass die geschlossenen Schalter deutlich niederohmiger sind als der Widerstand. Solche Widerstände zwischen dem Ausgang und VDD nennen wir PullUp Widerstände.

Also Out = !(A & B)

Folie 15 und 16

***

Wie realisieren wir jetzt die richtige UND Verknüpfung?

Hier brauchen wir einen Inverter.

Wir könnten es mit einem Schalter zwischen GND und dem Ausgang und einem Pull Up - Widerstand realisieren.

Wenn der Eingang hoch ist (=1), ist der Schalter geschlossen und der Ausgang ist niedrig.

Wir schalten also einen Inverter an NAND und bilden auf diese Weise UND Gate.

Folie 17

***

Auf ähnliche Weise können wir auch die Terme mit negierten Eingangsvariablen realisieren:

Z.B. Out = A & !B

Wir benutzen hier einen Inverter für b der Zwischen dem AND Eingang und dem Haupteigag für !B steht.

Folgende Symbole werden Verwendet:

Man kann den Inverter einfach durch einen Kreis darstellen. Zwei Kreise heben sich auf.

Folie 18

***

UND mit mehreren Eingängen

Folie 19

***

Auch „ODER“ Verknüpfung kann man mit Schaltern implementieren. Hier verwenden wir die Schaltern in Parallel. Die Schalter sind zwischen GND und dem Ausgang angeschlossen, wir benutzen einen PullUp Widerstand. Wir bilden zuerst NOR, dann hängen wir einen Inverter an.

Die Symbole sind auf der Folie dargestellt.

Folie 20

***

Unser Komparator benötigt also 256 UND Gatter mit jeweils 16 Eingänge und ein ODER mit 256 Eingängen.

Eine weitere Vereinfachung der Normalform nach den Absorptionsregeln ist in diesem Fall nicht möglich.

Was aber geht ist die Terme umzugruppieren. Dabei werden die Distributivregeln verwendet.

Folie 21

***

Wenn man logische Schaltungen entwirft, welche die Operationen zwischen Zahlen durchführen, baut man sie oft entsprechend der algorithmischen Methoden die man aus der Schule kennt.

Nehmen wir an, wir möchten zwei 8-Bit Zahlen vergleichen.

Wir würden die Zahlen bitweise vergleichen und wenn alle Bits gleich sind ist das Ergebnis auch gleich.

Die Schaltung für Bitweise Vergleich bekommen wir aus der einfachen Tabelle mit vier Zeilen.

Die Normalform ist

Y = (a & b) | (!a & !b)

Diese Funktion nennt man Äquivalenz. Die Inverse Funktion von Äquivalenz ist EXOR – Exklusiv ODER.

Folie 22

***

Ein 8-Bit Komparator basiert auf Bitvergleich ist in der Folie dargestellt.

Wir brauchen hier nur 8 x 2 UND Gatter und 8 ODER Gatter mit zwei Eingängen und ein UND mit 8 Eingängen.

Folie 23

***

Ein weiteres wichtiges Beispiel ist ein 8-Bit Addierer.

Auch hier bietet sich an, den gewöhnlichen Algorithmus für die Addition mehrstelligen Zahlen, den wir aus der Schule kennen Schaltungstechnisch zu implementieren.

Der Unterschied ist nur, dass wir es mit binären Zahlen zu tun haben.

Folie 24 und 25

***

Es ergibt sich folgende Tabelle für die Addition von zwei Bits a und b, wobei c der Übertrag aus der Addition des vorherigen Bits ist.

Die Gleichung für diese Addition ist:

Y = !c & (a exor b) | c & (a == b)

Wichtig ist auch der Übertrag

Cout = !c (a & b) | c & (a | b)

Folie 26

***

Die logischen Schaltungen, die die oben genannten Funktionen implementieren nennt man kombinatorische Logik. Der Ausgang der Schaltung ist definiert wenn man die Eingänge kennt.

In den digitalen Schaltungen verwendet man auch die Schaltungen mit Speicherelementen, mit denen man, zum Beispiel, zyklische Operationen durchführen kann, Zustandsautomaten baut oder Programme realisiert und durchführt.

Solche Schaltungen nennt man sequenziell, deren Ausgang hängt nicht nur von momentanen Eingangswerten sondern auch von der Vorgeschichte des Systems.

Folie 27

***

Ein Beispiel.

Nehmen wir als Beispiel eine Uhr – Timer.

Der Eingang ist die eingestellte Zeit – eine achtstellige binäre Zahl, und ein Start Knopf. Der Ausgang ist ein Signal das auf die abgelaufene Zeit aufmerksam macht.

Solche Systeme brauchen zunächst ein internes Taktsignal, also einen Oszillator.

Wir brauchen, weiterhin, einen Zähler. Den Zähler können wir mithilfe vom Addierer aufbauen. Wir brauchen zusätzlich einen Speicherelement in dem das Ergebnis gespeichert wird. Die Schaltung wird auf der Folie 28 gezeigt.

Folie 28

***

Das Ergebnis wird an A - Eingang des Addieres rückgekoppelt, am B-Eingang haben wir die feste Zahl 1. In einer Programmiersprache würden wir folgendes schreiben

A = A + 1

Folie 29

***

Die Frage ist nun wann und wie oft wird die Operation durchgeführt. Als Speicherelement wird üblicherweise ein Register benutzt. Dieses Register wird aus Speicherzellen aufgebaut – so genannten Flip Flips.

Folie 30

***

Die Flip Flops haben einen Eingang, Ausgang, einen Takteingang und oft ein Reset Signal.

Das Taktsignal hat eine periodische Form wie in der Folie dargestellt. Flipflops haben die folgende Eigenschaft. Der Wert am Eingang wird im Moment der steigenden Taktflanke gespeichert. Der gespeicherte Wert taucht auf dem Ausgang eine gewisse kurze Zeit danach, etwa ~ n x 100ps. So aufgebaut

funktioniert die Schaltung wie ein Zähler. Auf jede steigente Taktflanke erhöht sich der Zustand des Zählers.

Wichtig ist hier das folgende:

Folie 31

***

Der Eingang des Registers ändert sich auch einige 100ps nach der Taktflanke, da der Adierer den neuen Eingangswert A bekommt und seinen Ausgang anpasst. Diese Änderung der Addrierer-Ausgangs wird aber erst auf die nächste Taktflanke in Register gespeichert. Folgendes ist wichtig: die Änderung des Registerausgangs, verursacht durch Taktflanke i (Qi) führt zu einer Signalwelle durch die kombinatorische Logik. Diese soll vor der nächsten Taktflanke i+1 die Eingänge der nächsten Registerstufe erreichen als Zustand Qi+1 erreichen. Auf Taktflanke i+1 wird der Zustand Qi+1 gespeichert. Bei uns ist die nächste Registerstufe dasselbe Register.

Folie 32

***

In einer Hardware-Programmiersprache schreibt man

Always @ (posedge CLK) begin

A <= A + 1

End

Folie 33

***

Wie realisieren wir ein Flipflop?

Am einfachsten stellen wir uns eine Speicherzelle wie einen getakteten Kondensator vor. Wenn der Schalter geschlossen ist, verbinden wir den Eingang mit einem Kondensator. Der Kondensator wird auf das Eingangspotential aufgeladen. Erinnern wir uns dass die Kondensatoren Spannungen behalten,

wenn kein Strom aus ihnen fließt. Wenn der Schalter geöffnet wird, behält der Kondensator das Potential. Das logische Niveau wird auf diese Weise gespeichert.

Auf solche einem Prinzip funktionieren die DRAM Zellen.

Folie 34

***

Wenn wir solche Speicherzellen für den Zähler verwenden würden, gebe es ein Problem:

Nach der steigenden Taktflanke wird der Eingang gespeichert - das ist in Ordnung. Das Flip-Flop aus einem Kondensator würde aber jede weitere Änderung am Eingang ebenfalls speichern, bzw. das anfangs gespeicherte Wert überschreiben, solange Taktsignal eins ist. Das wollen wir nicht. Der Eingangswert soll nur auf die Flanke gespeichert werden, und der gespeicherte Zustand soll sich bis zur nächsten Talkflanke nicht ändern, auch wenn sich der Eingang ändert.

Wie realisieren wir dann eine Schaltung deren Zustand sich wirklich nur auf die Flanke ändert?

Folie 35

***

Eine Möglichkeit ist zwei DRAM Zellen hintereinander zu schalten. Die erste Zelle wird an negiertes Taktsignal und die zweite an Originaltakt angeschlossen.

Im Moment unmittelbar vor der steigenden Taktflanke ist die erste DRAM Zelle transparent, was heißt ihr Eingang (D1) ist mit dem Ausgang (Q1) kurzgeschossen. Die zweite Zelle ist vom Eingang D2=Q1=D1 getrennt und hält den alten Zustand Qi.

Nach der Taktflanke wird D1 in der ersten Zelle auf ihrem Kondensator gespeichert. Gleichzeitig geht die zweite Zelle in den transparenten Zustand so dass D1 auch an Q2 Hauptausgang sichtbar wird. Da der Eingang der ersten Zelle D1 jetzt vom Ausgang Q1 getrennt ist, haben weitere Änderungen am Haupteigang keinen Einfluss auf Q2.

Das gilt auch wenn das Taktsignal wieder auf null kommt. Dann wird zwar D1 mit Q1 kurzgeschlossen, gleichzeitig wird aber auch Q1=D2 von Q2 getrennt. Der zweite Kondensator hält den Zustand.

Beachten wir, dass die Schaltung nur dann richtig funktioniert wenn die zwei Schalter genau in Gegenphase sind. Jede Verzögerung könnte zu Fehlern führen. Z.B. es könnte passieren dass die zweite Zelle auf fallende Taktflanke überschrieben wird. Das Design von Flipflops soll vorsichtig gemacht werden.

Folie 36

***

Wir haben gesehen wie man den Zähler, seine kombinatorische Logik und Register mithilfe von Schaltern, Kondensatoren und Widerständen realisieren könnte.

In Wirklichkeit ist die Implementierung ein Bisschen anders, aber nach dem gleichen Prinzip wie die eben beschrieben.

Unser Timer braucht noch einige Elemente vor allen eine Zustandsmaschine, die, nachdem das Startsignal erzeugt wird, den Zähler resetet und startet. Wenn der Zählstand den vorgegebenen Wert erreicht, wird der Zähler von der Zustandsmaschine stoppt und ein Alarmsignal erzeugt. Diese Zustandsmaschine ist eine Art Hardwareprogramm und die eigentliche Intelligenz des Systems. Zustandsmaschinen sind, ähnlich wie Zähler, sequenzielle Schaltungen und werden auf die gleiche Weise aufgebaut. Wir werden uns damit in späteren Vorlesungen befassen.

Das Design von digitalen Grundkomponenten wie Flip Flops oder Gattern wird ein Thema dieses Kurses sein, obwohl es mehr im Gebiet analoge Elektronik liegt. Solche digitalen Komponenten werden heute ausschließlich auf den Chips als ICs implementiert.

Wenn man heute über Digitaldesign spricht, meint man hauptsächlich den Entwurf von digitalen Schaltungen auf einem FPGA oder einem ASIC. Die Schaltung wird dabei automatisch aus dem HDL Code in mehreren Schritten generiert. Im ersten Schritt erzeugt die Software eine Netzliste mit vorgegebenen Komponenten, welche dem Code entspricht. Im zweiten Schritt

wird ein Belegungsplan für die Komponenten erzeigt und die physikalischen Verbindungen auf dem ASIC, FPGA werden vorgesehen.

Im letzten Schritt werden die Verbindungen und die Schaltungen realisiert, indem z.B. das RAM des FPGA programmiert wird oder ein ASIC hergestellt wird.

Obwohl die digitalen Schaltungen aus dem Code automatisch generiert werden, ist es nützlich, die Realisierung zu kennen. Nur wenn wir ungefähr wissen welche Netzliste aus einem Code generiert wird, können wir die Performanz der Schaltung, wie die maximale Taktfrequenz, abschätzen und den Code entsprechend anpassen. Ein Beispiel. Wir hätten einen 10GHz 8 Bit Zähler. In dem Fall wäre es sinnvoll z.B. den Zähler auf zwei 4-Bit Zähler zu zerlegen, und den langsamen „MSB-Zähler“ an das 16x langsamere Taktsignal anzuschließen.