worum geht es in den nächsten beiden vorlesungen in den nächsten beiden vorlesungen wollen wir uns beschäftigen mit dem datenpfad wir haben ja in grundlagen der rechner architektur schon mal wo wir uns ein bisschen verschärft oder vertieft damit befassen. Um dieses Kapitel zu verstehen, wäre es gut natürlich, wenn Sie das vorausgegangene Kapitel ALU verinnerlicht haben. Es geht auch ein bisschen über digitale Schaltungen. Wir wollen ja wissen, wie sich einzelne Instruktionen auf die Hardware auswirken. Und wir werden am Ende des Kapitels dann auch nochmal über Pipelining sprechen. Worum geht es uns in diesem Kapitel? von drei verschiedenen Datenfahrtvarianten und unser Beispielprozessor ist der MIPS-Prozessor, der eben in der Literatur auch wirklich ausführlich und schön dargelegt ist. Das heißt, da kann man dann auch jederzeit im Hennessy-Patterson-Buch mal was nachlesen, wenn man was nicht verstanden hat. Gut, wir beschäftigen uns also erstmal mit dem Befehlsatz des MIPS-Prozessors und dann überlegen wir uns nochmal, was ein Taktschema bedeutet, kombinatorische und speichernde Elemente in so einem Datenpfad abwechseln und wie das getaktet werden muss und wir werden dann einführend besprechen einen sogenannten Single Cycle Datenpfad, das heißt wir gehen davon aus, dass ich in einem Zyklus einen kompletten Maschinenbefehl abarbeiten kann und alle Befehle werden gleich lang dauern bzw wir müssen nach dem Befehl schauen, der werden wir uns damit beschäftigen, wie ich diesen Single-Cycle-Datenpfad verändern muss, um einen Multi-Cycle-Datenpfad daraus zu machen. Das heißt, ich will mich ja nicht unbedingt nach dem Langsamsten richten, sondern eventuell eben, wenn ich unterschiedliche Befehlslaufzeiten habe, die dann auch als unterschiedliche Zykluszeiten implementieren. Und dieser Multi-Cycle-Datenpfad ist gleichzeitig Voraussetzung, um überhaupt so einen Pipeline-Datenpfad betrachten zu dann um die parallelisierung und wir wissen dann auch schon aus gra dass bei parallelisierung ist immer zu datenabhängigkeiten und konflikten kommen kann und wie man eben diese konflikte aus der welt schaffen kann darum geht es dann im rest dieses kapitels gut also die alle jetzt Das wurde alles aus dem MIPS R 2000 entnommen. Im Rahmen dieses Kapitels ist es synonym, ob ich jetzt Befehl sage, Maschinenbefehl sage oder Instruktion sage, es ist immer das gleiche gemacht. Und alle Beispiele kommen eben vom MIPS R 2000. jeden einzelnen dieser MIPS-Befehle sich mal in Ruhe anschauen. Und was ist dieser MIPS R2000? John Hennessy und seine Mitarbeiter an der Stanford University haben eben in den 80er Jahren begonnen, diese RISC-Architektur zu entwickeln, zu entwerfen und ja, das ist praktisch, wenn man so will, so eine der ersten richtig erfolgreichen RISC-Architekturen maßgeblich oder prägend für insgesamt die Risk-Architektur, wie wir sie heute kennen. Es gibt noch so eine andere Entwicklung von Hennessy und Patterson, das ist der sogenannte DLX-Befehlssatz und die beiden haben ja sehr eng zusammengearbeitet und im Großen und Ganzen ist es auch sehr, sehr ähnlich, also der DLX-Befehlssatz und der MIPSR2000-Befehlssatz. instruktionsverarbeitung mit integers beschränken und alle gleitkomma befehle wollen uns erst mal nicht interessieren gut der befehlszyklus ist ihnen ja im wesentlichen oder müsste ihnen im wesentlichen bekannt sein wir haben ja unsere fünf phasen schon frühzeitig definiert das ist instruction fetch instruction decode execute phase mem und writeback phase

diese und wirklich sehr genau bis ins letzte bit betrachten also die erste phase ist rock'n'fetch da geht es darum dass der befehls zähler praktisch im befehl speicher deutet auf den nächsten instruktion auf den nächsten befehl der auszuführen ist und somit dann diesen befehl zur verfügung stellt in der nächsten phase wird dieser befehl genommen und quasi auseinander genommen arithmetischen Befehl habe, stehen da drin zwei Operanten, die ich miteinander verknüpfen muss. Das heißt, ich habe zwei Read-Befehle für die zwei Operanten, dann berechne ich ein Ergebnis, dafür brauche ich einen Write-Befehl und wir haben Register-zu-Register-Befehle, das heißt, die Operanten kommen aus dem Register und das Ergebnis wird ins Register-File zurückgeschrieben. Die dritte Phase X, die macht praktisch diese Verknüpfung, also die Registerinhalte, die verarbeitet und der ausgang der alu wird zurückgeschickt in dieses register file so die ersten drei phasen sind im wesentlichen gleich für alle befehle die phasen 4 und 5 die sind jetzt ein bisschen unterschiedlich je nachdem welchen befehl man abarbeitet hier in grün gezeichnet ist die phase 4 sind für speicherbefehle das heißt unsere loadstore befehle also ich will etwas Da will ich Daten zurückspeichern. Und wir sehen das hier unten. Moment, jetzt kostet mir die Maus das wieder hin. Wir sehen das hier, diese grüne Bewegung. Also die ALU in der grünen Phase, in der vierten Phase, die berechnet mir eine Adresse. Nämlich die Adresse, wie ich auf den Speicher zugreife. Und dann je nachdem, wenn ich ein Store habe, dann will ich was in den Speicher schreiben. Dann kommen über diese Leitung hier unten meine Daten, die ich in den Speicher schreiben will. Oder wenn ich ein Load habe, dann kommen über diese grüne Leitung hier meine Daten aus dem Speicher und werden im Register-File und im Data-Eingang praktisch abgelegt. Das sind die zwei Möglichkeiten, die ich habe in der grünen Phase. Ich habe entweder ein Load oder ein Store. Die fünfte Phase, die letzte Phase, die wird als Write-Back-Phase bezeichnet. Die wird praktisch betrachtet für alle Befehle, die keine Load-Stores sind. Also zum Beispiel für arithmetische Befehle. kreislauf praktisch also ich habe hier meine operanten aus dem registerpfeil werden über die alu verknüpft und das ergebnis der alu wird dann über diesen roten pfad zurückgeschrieben ins registerpfeil also das ist ein echter daten kreislauf den ich hier habe geht von registerpfeil über alu zurück ins registerpfeil wenn ich arithmetische berechnungen ausführe was für Möglichkeiten habe ich, wie sieht die Architektur aus. Und erstmal sind hier definiert 32 Register für diesen Prozessor. Und zwar das Register 0 bis zum Register 31. Und klar, die sind in einem Registerpfeil organisiert und das sind eben, wenn man so will, schnelle Speicherorte für Daten, mathematischen und logischen Operationen. Es gibt so ein paar Spezialdefinitionen, das heißt das Register 0 entspricht immer der 0, also da steht immer die 0 drin und das Register 1 ist speziell reserviert für den Assembler, um zum Beispiel große Konstanten zu behandeln oder Pseudobefehle reinzuschreiben. Register 31, diese Register stehen dann mehr oder weniger zur freien Verfügung. Und dann gibt es noch das High und Low, das sind nochmal zwei 32-Bit-Register, die die Ergebnisse von Multiplikationen und Divisionen enthalten. Gut, dann haben wir unseren Speicher, wir haben ja eine 32-Bit-Architektur, das heißt unsere Adressen können erstmal maximal 32-Bit lang sein, Bit und so ein Speicher, wie Sie aus Grundlagen der Rechnerarchitektur gelernt

haben, das ist einfach durchnummeriert von 0 bis 4 Gigabyte, das ist ein Byte adressierter Speicher, das heißt, wenn ich ein Befehlswort betrachte, was ja 32 Bit sind, brauche ich immer 4 Bytes, also wenn ich um 1 Watt weiterschalten will, muss ich meine Adresse um 4 hochschalten. auch natürlich datenstrukturen arrays und spill registers was ich bei unterprogramm aufrufen bruch also der speicher steht mir sowohl für daten als auch datenstrukturen als auch befehle steht mir für alles zur verfügung so es gibt noch paar pseudo befehle für den MIPS assembler die müssen wir jetzt natürlich nicht alle auswendig lernen oder können einfach nur wenn jetzt der Name dieser Pseudo-Operationen, also Move und Clear und Not, Load Address und Load Immediate, Branch Unconditionally, Branch and Link, Branch If Greater Than, also Verzweigungen je nach Bedingungen, eine ganze Menge davon und dann so eine Multiplikation, die nur die unteren 32 Bit zurückliefert, ganz da unten, eine Division, die den Quotienten sind keine wenn man so will nativen mit befehle aber für diese befehle gibt es im mips befehl satz sequenzen die quasi diesen befehl abarbeiten und hier ist noch mal die eigentliche instruction syntax also wie man diese instruktion aufruft so hat ein move zum beispiel zwei register ein rt register geschrieben wird und das ist jetzt praktisch die real instruction also das ist die instruktion die tatsächlich im mips implementiert ist und wie sie aufgerufen werden muss ja also wenn ich so ein move machen will dann rufe ich praktischen ad auf von register rt und register rs und register zero und was ich letztendlich mache ich addiere null zum register zero dazu dann ist es natürlich das gleiche, als wenn ich den Inhalt von RS nach RT schreibe. Also das ist hier quasi die MIPS-Übersetzung für diese Befehle. Und hier können Sie in der Spalte die Bedeutung nachlesen und hier können Sie die MIPS-Befehlsequenz nachlesen, die ich brauche, um diesen Befehl umzusetzen. zur Verfügung gestellt habe. Also wenn es jemand ganz genau wissen will, dann kann er da quasi diese ganzen Befehle nachschauen. Und es ist nicht so, dass immer genau ein MIPS Befehl so ein Pseudo Befehl entspricht, sondern wir sehen es da unten, ab und zu brauche ich eben auch mehrere oder zwei MIPS Befehle, um so eine Anweisung umzusetzen. Also so ein Branch if greater than, da muss ich erstmal so ein meine Bedingungen auswerten und setzen. Und wenn die gesetzt ist, dann kann ich ein BNE, also ein Branch Not Equal ausführen und kann eben quasi IF RS größer RT, dann kann ich diesen Sprung ausführen. Also es wird hier übersetzt in zwei MIPS-Anweisungen. So ist diese Tabelle zu interpretieren. Wie gesagt, Sie müssen sowas natürlich nicht auswendig lernen, aber man sollte sich den einen oder anderen Befehl einfach mal anschauen, wie dieser MIPS Befehlsatz arbeitet. Gut, was jetzt auf den folgenden Folien kommt, sind die wichtigsten Befehle aus der MIPS Assembler Sprache, also das ist quasi die Zusammenfassung der wichtigsten Eigenschaften dieses kompletten Befehlsatzes und wir haben hier eben wie immer unterschiedliche Kategorien von Befehlen, also hier haben wir erstmal die arithmetischen Befehle, normale addition wir haben eine ganz normale subtraktion wir können add immediate heißt eine konstante irgendwie addieren ja wir haben also register 2 plus irgendeine konstante müssen wir natürlich darauf achten wie groß diese konstante maximal sein darf steht im handbuch dann drin wir haben ein add unsigned da wird praktisch eine vorzeichenfreie ganze zahl dazu addiert Add Immediate

Unsigned, also es sind jede Menge unterschiedlicher Additionen und Subtraktionsbefehle. Dann haben wir auch nochmal so einen Ausnahmebefehlszähler, den ich mir holen kann ins Register 1, das war diese Spezialbedeutung dann zur Not für dieses Register 1, was ich dann nochmal habe. Sätze, dann haben wir auch die Visionsbefehle hier, also das sind alles arithmetische Befehle, die hier ausgeführt werden und vom MIPS Befehlsatz entsprechend umgesetzt werden. Die zweite Kategorie von Befehlen sind unsere logischen Befehle, da haben wir also das logische und, das logische oder, wir können auch mit einer Konstanten vergleichen, mit und und mit oder, wir können einen Shift nach links machen, wir können einen Shift nach rechts Und beim Shift werden jeweils mit Nullen aufgeführt und wir können End oder natürlich Bitweise durchführen. Dann gibt es die Datentransferbefehle. Datentransferbefehle sind Load und Store. Ich kann also mit Loadword ein Wort aus dem Speicher laden und ich kann mit Storeword ein Wort in den Speicher zurückschreiben. Ich kann Konstante laden ins Register 1 in dem Beispiel und da muss ich auch wieder aufpassen, wie groß diese Konstante denn maximal sein darf. So, ich habe schon ganz kurz über diese 32 wichtigen Register gesprochen und hier ist nochmal die Konvention, wie diese Register beim MIPS belegt sein sollen, dürfen. MIPS Handbuch drin, dass niemand jetzt wirklich überprüft, ob man es tatsächlich so verwendet, diese Register, aber man sollte sich an diese Konventionen halten, weil sonst kann es eben zu unerwartetem Verhalten kommen oder eben auch zu Fehler kommen. Zum Beispiel das Register 0, da geht man einfach davon aus, dass es Hardwired mit 0 verbunden ist. Da steht einfach die 0 drin. Es hat auch dann keinen Sinn, was ins Register 0 reinzuschreiben, weil da wird immer die 0 drin stehen. Also ins register 0 zu schreiben macht keinen sinn register 0 wenn man so will ist einfach immer ein source register was ich dann verwende wenn ich die 0 brauche so auch das register 1 sollte man nicht unbedingt verwenden weil es eben für pseudo instruktionen verwendet wird und dann wenn da was drinsteht es einfach gnadenlos überschrieben wird also einfach mal finger weg vom register 1 und da gibt es jetzt ein paar Konventionen, das kann man sich dran halten, muss man sich nicht unbedingt dran halten, aber man muss halt aufpassen, dass dann keine Fehler passieren. Also Register 2 und 3 ist reserviert für Rückgabewerte von Funktionen, dann Register 4 bis 7 ist reserviert für Argumente, dann haben wir Register 8 bis 15 sind temporäre Daten, also die sind praktisch vollkommen frei nicht irgendwie vorreserviert von unterprogramm aufrufen dann haben wir die register 16 bis 23 die werden häufig bei unterprogramm aufrufen verwendet um eben aktuelle register zu retten dann haben wir register 24 und 25 das sind noch mal temporäre register die eben frei sind nicht weil jeder Kernel gerne darauf zugreifen möchte. Und dann haben wir Register 28, das ist so ein globaler Pointer. Dann haben wir den Stack Pointer in Register 29, den Frame Pointer in Register 30 und die Return Address in Register 31. Also wenn Sie mit Unterprogrammaufrufen arbeiten, dann sollte man eben beim Unterprogrammaufruf entsprechend diese drei Register mit Stack Frame Pointer belegen und mit der Return Address belegen. 32 Register, Integer-Register, die uns da zur Verfügung stehen, also 32 Bit-Register. Und darüber hinaus gibt es dann nochmal 32 Register, das sind die Floating-Point-Registers, wo also Gleitpunktzahlen irgendwie

abgelegt werden oder Ergebnisse von Gleitpunktzahlen-Berechnungen abgelegt werden. Diese Register wollen wir jetzt in dieser Vorlesung nicht betrachten, da wir auch jetzt keine Gleitpunktbefehle betrachten. dass der MIPS eben auch über Gleitpunktregister verfügt. Gut, wir haben eine Kategorie noch nicht betrachtet, das sind unsere Sprungbefehle. In der Kategorie Sprungbefehle, da gibt es jetzt die Unterscheidung zwischen einem Sprung mit Bedingung und einem unbedingten Sprung. Der Sprung mit Bedingung heißt immer Branch und je nachdem wie ich diese Bedingung auswähle, Also Branch on equal, das heißt ich springe, wenn die zwei Argumente, die ich übergebe, gleich sind. Branch on not equal, ich springe, wenn sie nicht gleich sind. Ich habe ein Set on less than, also da setze ich meine Bedingung, wenn Operand 2 kleiner ist als Operand 3. Set on less than immediate, hier vergleiche ich mit der Konstanten und wenn eben dieses Dollar 2 kleiner ist als die Konstante, dann wird diese Bedingung in Dollar 1 gesetzt hier. Und set less than unsigned, da vergleiche ich eben bei unsigned, das sind immer vorzeichenfreie Zahlen, das heißt natürliche Zahlen, da gehe ich davon aus, dass dieses Dollar 3 eben eine natürliche Zahl ist. So und Auswertung ist klar, wenn Dollar 2 kleiner Dollar 3, dann wird die Bedingung auf true gesetzt in Dollar 1 und ansonsten wird sie eben auf 0 gesetzt, so wie bei den anderen vorher auch. Set on less than immediate unsigned, also auch hier der Unterschied von hier zu hier ist einfach, dass hier immer davon ausgegangen wird, dass dieses Argument, diese Konstante, dass das eine natürliche Zahl ist, das heißt da sind keine negativen Zahlen erlaubt, während hier, wenn nur immediate steht bei der Konstanten, dann wäre auch eine negative Zahl erlaubt. Das heißt, das ist jetzt die Adresse des nächsten Befehls, der ausgeführt wird. Ich springe also zu dem Befehl an der Stelle 10.000. Oder ich springe an die Adresse, die in einem Register hinterlegt ist, also Jump Register. Das heißt, das, was im Register 31 hinterlegt ist, das ist praktisch die Befehlsadresse, die als nächstes aufzurufen ist. Und für Prozeduraufrufe gibt es noch so einen Jump & Link mit JAL abgekürzt, 10.000. Und da wird praktisch dann dieses Register 31, das ich eben für Unterprogrammaufrufe verwende, was da drin steht, wird gesetzt mit dem Befehlsteller plus 4, also mit dem nächsten Befehl praktisch. Das heißt hier wird der Befehlsteller nochmal gesetzt und anschließend springe ich an die Stelle, die hier in der Emilia hinterlegt ist, nämlich an die Stelle 10.000. Das sind also quasi zwei Befehle, die hier ausgeführt werden. Gut, Sie erinnern sich noch an das Kapitel, wo wir über Instruction Set Architecture gesprochen haben und so quasi auf der letzten Folie war ja die Quintessenz dieses Kapitels, war genau dieses Bild hier und das gilt auch für den MIPS Prozessor im Speziellen. Das Format I, das sind Transferbefehle und Befehle mit irgendwelchen Konstanten. Und das Format J steht für Sprungbefehle. Und es gilt natürlich, die 32 Bits, die mir zur Verfügung stehen, aufzuteilen vernünftig auf jedes dieser Befehlsformate. Für alle Befehlsformate gilt, dass die obersten 6 Bit, da steht der Opcode drin. Also welche Operation ist auszuführen. der Code für die Operation Addition, Subtraktion oder logisches Und. Und hier beim Format I-Befehl, wenn es ein Transferbefehl ist, wird zum Beispiel die Codierung für Load oder Store drinstehen und hier eben der entsprechende Sprungbefehl informatiert. Dann haben wir für die nächsten 5 Bit, für die Format R und

I-Befehle, da drin steht, ja das ist also bei den arithmetischen Befehlen ist es der erste Operant, der da drin steht und auch bei den Format I Befehlen steht da ein Operant praktisch drin, das sind also Quellregister, hier haben wir auch nochmal Quellregister stehen, meistens sind das Quellregister je nach Befehl, kann das beim Format I auch was anderes drin stehen, aber normalerweise steht da Quellregister drin. praktisch und bei den transfer befehlen steht da die adresse drin auf die ich zugreifen will oder irgendeine konstante drin und dann haben wir noch so eine shift amount für die arithmetischen befehle dann kann ich noch mal in 6 bits kodieren wie viel ich shiften will und wie viel ich shiften will und diese entschuldigung das sind 5 bits und diese funct 6 bits da steht jetzt noch mal ergänzende Operationsinformationen drin. Da werden wir dann im Laufe dieser Vorlesung noch sehen, für was wir diese Bits hier hinten brauchen. Gut, beim Format J-Befehl, da habe ich nur zwei Teile in meinem Register. Da steht erstmal die Operation in den oberen 6 Bits und hier steht eben die Adresse drin. Also ich kann bis zu 28 Bits für die Target-Adresse verwenden. Das ist also das Maschinensprachenformat für den MIPS und wir wollen uns jetzt im Verlauf der nächsten beiden Vorlesungen wirklich genau anschauen, wie die Belegung dieses Registers, dieser Befehle praktisch mit Nullen und Einsen, wie sich das konkret auf die Hardware auswirkt. wie der kodiert ist da sehen wir das also für die operation für die ersten sechs bits da steht da die null drin also das stehen jetzt hier dezimalzahlen drin die natürlich dann in bit muster umgewandelt werden hier die funktion addition ist praktisch hier in dieser funkt kodiert das ist die zahl 32 bei 32 weiße er soll ein ganz normales app machen und hier das sind register 3 drin register 1 drin und nicht geschifftet also ist shift amount natürlich gleich null für das add und sie sehen es gibt ja nicht nur ein add es gibt auch ein add integer und oder ein add unsigned und da sehen sie dass ich das eben hier in der belegung da vorne unterscheide entschuldigung nicht add integer sondern add immediate ist es natürlich hier sehen kann ich ja hier in funct das nicht kodieren, also diese Bits stehen mir nicht zur Verfügung hier und deshalb ist das hier in den ersten OP, in den ersten 6-Bit eben dieses addImmediate kodiert. Beim addUnsign steht da vorne wieder die 0, ich habe jetzt hinten wieder meine Bits für die Funktion und das normale add hat also die Nummer 32 und das addUnsign hat die Nummer 33. Also über diese letzten 6 Bits wählen Sie aus, welche Addition, welche Subtraktion, welche logische Operation jetzt genau gemacht werden soll. Und die vorderen Bits brauchen Sie eben zum Beispiel bei Operationen mit Konstanten. Das ist also nur eine Beispiel-Tabelle. Das sind natürlich nicht alle MIPS-Assembler-Befehle, die hier abgebildet sind. Das sind ein paar von den wichtigen Befehlen. muss dafür einen maschinenbefehl aussehen kann hier haben wir noch ein beispiel für logische operationen für sprung operationen also end immediate or immediate das sind logische operationen mit konstanten das sind shift left shift right das sind shift operationen da brauchen wir jetzt natürlich die shift amount größe hier dann wenn wir shift operationen haben immediate und dann haben wir hier die Sprungbefehle mit branch unequal, branch not equal und dann haben wir hier die setzenden Bedingungen, set unless then, set unless then immediate, set unless then unsigned, set unless then immediate unsigned. Das sind also so feine

Unterscheidungen in den einzelnen Befehlen, die sich natürlich in der Belegung der Bitfilter hier entsprechend widerspiegeln. mal Jump & Link Befehl hier unten und das ist eben hier vorne auch noch mal kodiert in der OP Nummer, welchen dieser Sprünge Sie jetzt ausführen wollen. Gut, was ist jetzt unser Ziel? Was wollen wir eigentlich machen? Wir wollen jetzt erstmal einen simplen Datenpfad konstruieren, wo eben dieses Befehlsformat oder diese Befehlsformate unterstützt werden. die unterscheiden sich ja nur im Detail. Das heißt, die wichtigsten Befehle, das ist der Speicherzugriff, also wie führe ich einen Load aus, wie führe ich einen Store aus. Dann haben wir als wichtigste Befehle in der Addition, Subtraktion, die und-logische, und-logische oder und die Set-Less-Then-Operation. Und dann haben wir noch eben zwei Arten von Sprüngen, bedingte Sprünge oder unbedingte Sprünge, das heißt Branch-Equal oder Jump. das Ziel bei der Entwicklung eines Risk Reduced Instruction Set Befehlssatzes ist, dass es möglichst einfach, möglichst regelmäßig ist. Das heißt, wir haben nur drei unterschiedliche Befehlsformate und dementsprechend unterschiedliche Belegungen innerhalb eines Befehls, damit ich eben daraus aufbauend eine einfache Hardwarestruktur adressieren kann. Außerdem gibt es ein paar Das heißt, die häufigsten Fälle müssen die schnellsten sein und eben Einfachheit erlaubt Regelmäßigkeit, also lieber einen komplexen Befehl in zwei einfache Befehle übersetzen, als meine ganzen Formate und meine ganzen Hardware auf einzelne komplexe Befehle anzupassen. für false steht, also für ein inaktives Signal und dass die 1 immer für true steht, also für ein aktives Signal. Das heißt, wenn wir so einen Schaltplan sehen und auf einer Leitung steht eine 1, dann heißt das eben true und wenn da eine 0 steht, dann heißt das eben false. Das ist Definitionssache und wir definieren es einfach so. Gut, jetzt müssen wir uns nochmal ganz kurz über die Taktung unterhalten, weil die natürlich essentiell Datenpfad. Wir erinnern uns ganz grob ans Kapitel Automaten in der Grundlagen- und Rechnerarchitektur, da wurde das ja erstmalig vorgestellt und so ein Prozessor ist ein sequenzieller Schaltkreis und ein sequenzieller Schaltkreis kann immer aufgeteilt werden. Ich habe einmal speichernde Elemente, also das sind so die Zustände, wenn Sie sich erinnern, bei den Automaten und die kombinatorischen Elemente, das sind die Verknüpfungen, die da entstehen. Das enthält den Zustand, also Register, das sind Speicherelementen, Datenspeicher, Befehlsspeicher. Wir haben ja hier immer einen Speicher, wo wir die Instruktionen rausholen am Anfang unseres Datenpfades und am Ende des Datenpfades steht praktisch der Datenspeicher, wo wir unsere Operanten holen und unsere Ergebnisse abspeichern. In der Hardware werden solche Speicher In dem Beispiel hier eben so ein D-Flip-Flop. Und was passiert jetzt mit dem Takt? Der Takt bestimmt praktisch den Zeitpunkt. Ich habe hier beim D-Flip-Flop einen Eingang, dieses D. Ich habe einen Ausgang, dieses Q. Und wenn ich das Ganze takte hier mit einem Takteingang, der heißt hier Clock, dann wird der Zeitpunkt bestimmt, an dem die Information vom Input D auf den Output Q übertragen wird. Punkt kann ich dieses Q auslesen. Nur wenn ich takte, dann wird eben dieses D durchgeschaltet zu dem Q. Und das Ganze geschieht in unserem Beispiel hier immer bei der steigenden Taktflanke. Das heißt, mein Takt hat ja erstmal so eine steigende Taktflanke, dann bleibt er oben, dann wird er wieder runtergezogen. Hier passiert nichts bei der fallenden

Flanke und bei der nächsten steigenden Flanke wird der Takt erneut gegeben und dann wird erneut ausgelesen. Zeitpunkt wird das D auf Q durchgeschaltet und zu diesem Zeitpunkt wird das D auf Q durchgeschaltet. Und dazwischen kann der Ausgang Q praktisch jederzeit ausgelesen werden. Der ist dann natürlich immer konstant. Also das heißt, das Taktzyklus-Schema bestimmt die Sequenz von Zeitpunkten zum Lesen und Schreiben meines D-Flipflops. und nur bei steigenden Taktflanken. Es ist andere Möglichkeiten, dass man sowohl den steigenden als auch den fallenden Takt nutzt, aber das wollen wir eben hier nicht betrachten. Und außerdem gibt es noch eine andere Alternative, dass ich Zustandsgesteuere Flipflops nehme, also immer wenn sich ein Zustand ändert, ändert sich mein Flipflopwert. Auch das wollen wir nicht betrachten, sondern wir gehen von einem festen Takt Schema aus. ganz gut verwenden für unsere single cycle konfiguration das heißt wir haben hier ein storage element wir haben hier also ein speicher wir haben hier unsere kombinatorik und dann haben wir hier den nächsten speicher und wir haben hier unsere takteingänge bei diesen speicherelementen und wir haben eine bestimmte laufzeit die es dauert unseren schaltkreis hier zu durchlaufen und alle Gatter müssen durchlaufen werden von Anfang bis Ende, das soll diese Zeit TD symbolisieren, das ist die Gatterlaufzeit. So, und es ist jetzt klar, wenn ich jetzt den Ausgang Q hier auswerte und als Eingang in meine Kombinatorik nehme, dann muss ich natürlich warten, bis diese Kombinatorik berechnet hat, dann liegt er hier am Eingang des nächsten Speicherelements an und ich darf diesen Eingang erst dann übernehmen, wenn praktisch diese Kombinatorik durchlaufen ist. erst kommen, wenn genügend Zeit von hier nach hier vergangen ist. Also hier haben wir die erste Taktflanke, da wird der Ausgang hier ausgewertet und bei der zweiten Taktflanke steigenden, da wird der Eingang hier ausgewertet. Das heißt, die steigende Flanke ist die aktive und zu diesem Zeitpunkt wird eben das eingehende Signal gespeichert und die Taktperiode T, die muss natürlich mindestens so lang sein wie dieses Td. dimensionieren damit die Wartezeit lang genug ist. So was passiert jetzt wenn meine Kombinatorik länger dauert das heißt wenn mein T kleiner ist als das Td dann habe ich natürlich immer noch die Möglichkeit dass ich einfach zum auswerten Takte auslasse das heißt wenn die Kombinatorik mehr als einen Taktzyklus dauert dann muss ich halt muss dann die entsprechende Taktflanke zum Auswerten nehmen. In dem Beispiel hier gezeigt, also wir haben zum ersten Takt, da werten wir den Ausgang vom ersten Speicher aus, dann lassen wir den durch die Kombinatorik laufen, hier kommt der zweite Takt, da wird das Ergebnis noch berechnet, liegt noch nicht vor und erst beim dritten Takt bin ich fertig und dann verwende ich halt hier den dritten Takt, um diesen Eingang hier zu übernehmen und das Ergebnis zu übernehmen. Ich muss dann halt ausrechnen, wie viel Glatter habe ich, wie lang dauert das TD, wie lang dauert mein Takt, muss dann eben den entsprechenden Zähler einsetzen, damit der richtige Takt genommen ist und Bedingung ist natürlich immer, dass die Ausgabe des Speicherelements eben bis zum dritten Takt stabil anliegt, damit ich das korrekte Ergebnis übernehme. Gut, ich kann natürlich den Ausgang der Kombinatorik kurz schließen mit dem Eingang meines Speicherelements. Und das ist ja das, was in der Regel geschieht. Hier ist nochmal das symbolisiert als Ausschnitt

aus unserer Architektur. Wir haben hier unser Registerpfeil, das ist unser speicherndes Element. Wir haben hier unsere ALU, das ist das kombinatorische Element. Und genau diesen Kreislauf beschreiben wir, dass wir also aus dem Registerpfeil unsere Eingänge für die Kombinatorik bestimmen und das Ergebnis der Kombinatorik dann ins Registerpfeil zurückspeichern. Konfiguration liegt bei uns vor und das ist auch für Taktflankengesteuer Flipflops zulässig. Ich muss meine Taktheit entsprechend dimensionieren und wenn eben die aktive Taktflanke kommt, dann wird die stabile Ausgabe aus der Kombinatorik in das Speicherelement übertragen. Das Ganze haben wir eben schon mal besprochen im Kapitel Endliche Automatengrundlagen Rechnerarchitektur. haben wir immer so ein speichernes Element, wir haben Kombinatorik und wir haben unseren Taktanliegen, der eben bestimmt, wann die Ein- und Ausgänge ausgewertet werden. Also diese speichernen Elemente und kombinatorischen Böcke, die wechseln sich ab und Synchronisierung erreiche ich durch den Takt. Gut, jetzt wenden wir das mal alles an und bauen daraus einen Befehlszyklus mit einem Single-Cycle-Daten-Takt. Also es hat sich natürlich nichts geändert an unserem Datenpfad an sich. Wir haben nach wie vor unsere fünf Phasen. Instruction Fetch, Instruction Deco, Execute Phase, Mem Phase und Write Back Phase. Diese Grundsequenz, das ist genau die, die wir schon vorgestellt haben und die, die wir betrachten wollen. Vielleicht noch mal ein kleiner Blick drauf. Wir haben hier eine Harvard-Architektur vorliegen. Wir haben einen getrennten Speicher für unsere Befehle, das ist unser Instruction Storage hier vorne und einen getrennten Speicher für unsere Daten. Das ist hier unser Datenspeicher, wo die Load-Store-Operationen drauflaufen. Hier haben wir unser Register-File und hier haben wir unsere ALU, also ganz genau so, wie wir das eigentlich schon kennen. Was bedeutet jetzt Single-Cycle-Datenfahrt nochmal? Datenpfad werden alle Operationen dieser Grundsequenz, also alle fünf, Instruction, Fetch, ID und so weiter, während eines Taktzyklus ausgeführt. Das heißt, ich muss meinen Takt so lang dimensionieren, dass wirklich die komplette Grundsequenz ausgeführt werden kann. Jetzt muss man natürlich irgendwie ausrechnen, wie lang dauert denn dann dieser Takt. Dafür müssen wir wirklich alle einzelnen Befehle mal betrachten und uns dann eben den langsamsten Befehl suchen. Schritt für Schritt im Detail die einzelnen Phase und Dauer praktisch der einzelnen Phase versuchen abzuschätzen bzw. zu berechnen. Also die erste Phase, Teil-Daten-Fat-Instruction-Fetch, die ist für alle Befehlstypen gleich. Ich muss mir ja erstmal den Befehl holen und es ist vollkommen egal, welchen Befehl ich hole, ob das jetzt ein arithmetischer Befehl ist oder ein Transferbefehl, er muss erstmal geholt werden. in unserer Architektur. Ich brauche den Befehlsspeicher, da steht er drin. Ich brauche den PC, den Program Counter, das ist praktisch der Pointer auf die Adresse, wo der Befehl jetzt steht. Und ich brauche einen Addierer, weil nach jedem Befehl ist der nächste Befehl. Das heißt, ich muss automatisch meine Befehlszelle dann auf den nächsten Befehl setzen, damit ich in der nächsten Befehlsholphase den richtigen Befehl hole. In Register Transfer Operationen ausgedrückt heißt es, ergibt sich aus dem instruktionsspeicher an der stelle pc und der program counter wird um 4 erhöht so das heißt also wir haben hier ein register für unseren program counter der darin steht aus diesem register ergibt sich die adresse mit der ich auf

instruktionsspeicher zugreife hier ergebnis ist die instruktion die dann praktisch in ein instruktionsregister umgeladen wird ja Diese ALU ist relativ langweilig, die addiert immer 4 auf den Befehlszeller drauf, damit er eben auf dem nächsten Befehl dauert. Gut, dieser Teildatenpfad, UIFatch, identisch für alle Befehle. So, jetzt müssen wir ein bisschen unterscheiden. Die nächsten Phasen sind eben nicht ganz identisch für alle Befehle. Da haben wir erstmal den Teildatenpfad für die R-Befehle, also für arithmetische und logische Operationen. wie sieht es so aus? Also wir haben hier aus unserer iFetch-Phase die Instruktion anliegen. In dieser Instruktion kodiert, enthalten, sind praktisch die Adressen für die zwei Inputregister, Register 1 und 2 und die Adresse für das Ergebnisregister. So, das heißt, wir haben erstmal zwei Leseregisteradressen, das sind die Inputs und eine Schreibregisteradresse. aus der Instruktion vor. Das sind diese drei Adressen, die anliegen. Dann haben wir hier eine Datenleitung, die mir das Ergebnis von der ALU liefert und hier praktisch hier werden die Daten abgeliefert und dann eben im Register-File unter der richtigen Registeradresse gespeichert. Diese Adresse liegt ja hier an, die steht ja hier im Write Register drin. Wir haben also einmal die Schreibdaten, die kommen hier als Input an, das ist das Ergebnis, das ankommt. Dann haben wir zwei output das sind die hier wir haben hier die adressen und wenn ich die adresse habe dann wird hier am ersten output meines register files der erste operand angelegt und am zweiten output der zweite operand so und dann brauchen wir natürlich ein kontrollsignal write wann diese information hier übernommen werden soll und außerdem müssen wir der alu natürlich mitteilen über steuerleitungen ausgeführt werden soll wir brauchen also mindestens drei bits weil wir wollen hier ein beispiel fünf operationen erst mal betrachten und fünf operationen kann ich in minimal 3 bit kodieren gut also vorgehensweise ist klar die instruktion liefert mir die ganzen adressen die ich brauche und dann habe ich einmal so ein feedback zyklus das heißt hier habe ich meine kombinatorik hier habe ich meinen meine feedback leitung so wie sieht jetzt der teildatenfahrt aus die überlegung stimmen ein bisschen öl wie sieht der teildatenfahrt aus für lade und speicher befehle schauen wir uns mal so ein lot wert an das ist hier gelb hinterlegt gelb gekennzeichnet also ich greife auf den speicher zu mit einem Adresse und das was ich lese lade ich ins Registerfile zurück. Wir haben hier wieder unsere Instruktionen die mir die wichtigen Informationen wie sie hier gelb hinterlegt sind senden. Das heißt mein Registerfile stellt mir hier oben bei Read Data 1 stellt es mir praktisch das $2 zur Verfügung. Der Offset kommt hier unten über Design Extension. Der Offset ist aufgrund des befehlsformats das heißt ich muss den vorzeichen richtig auffüllen zu 32 bit weil meine alu verarbeitet nur 32 bit operanten und dieser offset wird hier genommen zusammen mit diesem dollar zwei um ja wir haben hier eine addition stehen um die adresse auszurechnen das heißt das ergebnis der addition beim load ist in dem fall unsere read adresse ja das wird also am unter welcher Adresse ich zugreifen will, dann habe ich ein Load. Load heißt, ich will mir Daten aus diesem Datenspeicher holen. Ich habe also hier meinen Redata-Ausgang und der wird praktisch dann eben beim Load hier zurückgeleitet auf dieses WriteData und damit wird es also im Registerpfeil abgelegt, unter welcher Adresse. Das haben wir ja ausgerechnet. Beziehungsweise die ist ja hier mit Dollar 1 gegeben, für das

Registerpfeil. adresse vom speicher vom datenspeicher und das ist die adresse im register feile licht auch also so kann ich mein load aber bei wie schaut es aus beim store das schaut ganz ähnlich aus wie beim load also hier oben liegt praktisch an die adresse die zu berechnen ist über ein offset kommt praktisch der zweite teil kann also hier wieder meine adress berechnung ausführen das ist ja genau das gleiche wie beim load damit liegt dann die adresse an das ist aber jetzt anlegt und dann muss ich jetzt noch wissen ja welche daten werden denn praktisch geschrieben es wird das geschrieben was im register dollar 1 steht das heißt ich leite das was im dollar 1 ist an diesem ausgang read data 2 und dieser ausgang wird praktisch auf den eingang des datenspeichers gelegt und dann kommt hier eben die adresse die ich brauche wenn ich ihn auf dem speicher zugreife und kann dann das Signal geben, das Write-Signal, jetzt soll geschrieben werden, dann wird praktisch der Eingang übernommen und unter dieser Adresse in den Speicher geschrieben. Sie sehen also, so kann ich relativ einfach ein Load und ein Store realisieren mit unterschiedlichen Adressen. So, dann haben wir noch das dritte Format, was wir betrachten müssen, das ist nämlich dieses Sprungformat. also unser Befehl, den wir hier im Beispiel betrachten wollen, ist ein Branch-Equal $1, $2 Offset. Das heißt, was muss ich machen? Ich muss erstmal das Sprungziel berechnen und das Sprungziel ist der Program Counter plus das Offset, auch wieder mit einer Sign-Extension, also mit einer vorzeigenrichtigen Auffüllung, weil der Offset ist ja wieder nur 16-Bit-breit und ich brauche 32-Bit. sprungziel berechnet habe dann kann ich jetzt meine bedingungen auswerten also wenn dollar 1 gleich dollar 2 dann springe ich dann ergibt sich also mein programm counter aus dem sprungziel und ansonsten springe ich nicht und wenn ich nicht springe dann heißt es einfach dass mein programm counter um 4 erhöht wird ja programm counter ist 32 bit breit und wenn ich den nächsten an die nächste beit adressiert der speicher deshalb muss ich vier beides weiterschalten gut aufpassen dass vorher ja schon mein programm kommt inkrementiert wurde und ich also hier nochmal im krimi mit ihr krimi tiere wenn ich hier das sprungziel berechnen und deshalb muss ich das um zwei mit dieses offset um zwei bin nach und speicher byte interessiert sind also dieses offset ist ein word offset ich muss die richtige adresse berechnen und von word auf byte ist eben die multiplikation mit 4 und deshalb muss ich das um 2 bit nach links verschieben so das sehen wir also hier in dieser schaltung wir haben hier der hier reinkommt mit seinen extension eben vorzeige richtig aufgefüllt wird zu 32 bit und bevor wir diesen offset verarbeiten ja machen wir ein shift left um 2 eben deshalb weil der offset eine word adresse sind wir aber eine byte adresse brauchen also multiplizieren wir das ganze mit 4 und wir haben hier noch mal eben den alten teil ja dann haben wir den program handler aus anliegt und hier berechnen wir das sprungziel und wenn ich springe dann wird als neue adresse das genommen was hier aus diesem die er da rauskommt ja das wird dann durchgeschaltet als neue program counter und wenn ich nicht springe dann nämlich diese abzweigung hier und neben den vorher berechneten programm kaum der nicht schon ausgerechnet habe und schaltet den durch ich durchspeichern so was macht jetzt der teil hier unten der teil hier unten der wendet mir die bedingungen aus ich muss ja noch feststellen ja springe ich jetzt oder springe ich nicht und wir machen ein

branch equal also ich springe wenn beide gleich sind das heißt was mache ich ziehe die beiden voneinander ab dollar 1 minus dollar 2 und wenn dann null dabei rauskommen dann waren sie gleich dann setze ich also hier das zero und dieses ergebnis des vergleichs das muss ich natürlich noch irgendwie das ist jetzt hier nicht eingezeichnet über also ich muss ja dementsprechend dann diesen Multiplexer hier ansteuern, diese Ansteuerung ist noch nicht eingezeichnet, aber das ist eben die sogenannte Sprungkontrolllogik, die wir später betrachten wollen und was ich nur machen muss, ist dieses Ergebnis an die Sprungkontrolllogik weiterleiten, dann weiß ich, ob ich springen muss oder nicht springen muss. Gut, so oder so ähnlich funktioniert es natürlich für alle Sprungbefehle. Wir können jetzt nicht jede Ausprägung dieser Sprungbefehle betrachten, können Sie sich vorstellen, welche Komponenten beteiligt sind und was diese Komponenten im Wesentlichen zu tun haben bei dem Sprungbefehl. So, jetzt wollen wir diese ganzen Teildatenpfade, also die Instruction-Fetch-Phase und dann die anderen Phasen sowohl für arithmetische Befehle als auch für Transferbefehle, also auch für Sprungbefehle, die wollen wir jetzt alle zusammenbauen. Datenpfad zusammenbauen und jedes Element, das ich hier in diesem Pfad habe, darf natürlich maximal einmal pro Taktzyklus verwendet werden, weil wir haben ja einen Takt, wo wir alles für einen Befehl berechnen wollen und dementsprechend muss ich also diese separate Befehls- und Datenspeicherarchitektur unbedingt beibehalten und wenn ich also unterschiedliche Speicher Ich muss ja ab und zu zwischen Leitungen hin und her schalten und immer wenn das der Fall ist, also wenn ich wählen muss zwischen unterschiedlichen Eingabeinformationen, dann baue ich da natürlich Multiplexer rein, die ich dann über Kontrollleitungen ansteuern muss. Und wir wollen uns also jetzt anschauen, die notwendigen Erweiterungen, die wir treffen müssen, um diese ganzen Teildatenfahrte kombinieren zu können. vom register file kommt oder von der konstanten kommt und wir brauchen einen multiplexer für register write das eben wählt zwischen dem ergebnis was die alu berechnet hat oder daten aus dem datenspeicher ja um einfach zu unterscheiden zwischen load befehlen und arithmetischen befehlen wir sprechen von diesen beiden multiplexern hier und der multiplexer der ist eben dafür zuständig zu unterscheiden ob jetzt die konstante ob ich Read Data 2 hier als Eingang für die ALU nehme. Und der Multiplexer entscheidet praktisch, ob das Ergebnis aus der ALU hier weitergeleitet wird aufs Registerpfeil. Das wäre dann der Fall, wenn ich eine arithmetische oder eine logische Operation habe. Oder ob das Ergebnis aus dem Speicher hier weitergeleitet wird ins Registerpfeil. Das ist genau dann der Fall, wenn ich eine Load-Operation habe. einen Datenpfad für Loadstore und für R-Type Operationen, also für Transfer und R-Type Operationen zu zeichnen. So, der Hals wird trocken, müssen wir doch ein bisschen was nehmen. auch natürlich noch einbauen für einen Sprungbefehl. Sie erinnern sich, vorhin haben wir ja das berechnet, dass wir hier diesen Shifter brauchen, wegen Byte- und Wortadressierung, dass wir hier diese ALU brauchen, um unser Branch-Target-Adress, also um unser Sprungziel zu berechnen und wir müssen natürlich dann entsprechend umschalten. Über diesen Multiplexer wollen wir die Adresse vom Sprung oder wollen wir die reguläre Adresse, PC zu legen. Also, diese roten Teile hier, das sind die Erweiterungen, die wir brauchen, um

auch Sprungbefehle behandeln zu können. Und wenn wir diese roten Teile auch noch einbauen, dann sind wir jetzt in der Lage, mit unserem Datenpfad alle drei Befehlsformate anzusprechen. Also R-Type Befehle, I-Type Befehle und eben auch die Branch Befehle. gut so das ist im wesentlichen der datenpfad den wir brauchen was wir jetzt betrachten müssen was wir einfach überhaupt noch nicht drin haben ist wie steuern wir denn unsere multiplexer an wie steuern wir unsere alu an also wir haben noch keine kontrolleitungen wir haben wenn wir so wollen einfach nur den datenfluss uns jetzt mal aufgezeichnet das heißt die alu hat jetzt eben steuereingänge wir haben nur fünf wir betrachten momentan nur fünf mit drei bits kann ich ja acht kombinationen belegen wir wollen aber eben nur betrachten addition subtraktion logisches und oder und das set less than das heißt fünf operationen drei leitungen und wie wir denn die allo überhaupt genutzt ja wenn wir ein load store befehle haben dann nutzen wir die allo um eine adressberechnung Wenn ich ein Branch habe, dann benutze ich die ALU, um den Wahrheitswert zu berechnen, also um die Bedingung auszuwerten. Wir haben ja vorhin gesehen, wenn wir sehen wollen, ob zwei Operanten gleich sind, dann ziehe ich die voneinander ab und wenn das Ergebnis 0 ist, dann waren sie gleich. Also dafür brauche ich eine ALU für dieses Abziehen. Sie erinnern sich ans Kapitel vorher, wo wir über Alu-Design gesprochen haben, über diese Kontrolleingänge an der Alu, wo ich eben umschalte zwischen Addition, Subtraktion, logischen Und, Oder und dem Set Lester. Das baut natürlich klar auf die Betrachtungen des Kapitels vorher auf. und wir bauen eine kleine Logik für unsere ALU-Control. Wir haben ja gesagt, wir wollen unsere ALU mit 3 Bits ansteuern und wir haben aber insgesamt natürlich aus dem Maschinenbefehlswort, haben wir im Funktionsteil 6 Bits, die uns zur Verfügung stellen und aus dem Operationsteil könnte man auch 6 Bits verwenden, da wollen wir aber nur 2 Bits verwenden. Also wir verwenden 2 Bits aus dem ALU-Operationsteil um eben letztendlich unsere fünf operationen zu adressieren für diese fünf operationen brauche ich aber nur drei kontroll leitungen und diese alu-kontroll ist eine logik die mir praktisch diese 8 bits umsetzt in diese drei bits und wir definieren einfach folgendes wir sagen okay wenn hier oben bei alu-op wenn da die 00 anliegt dann will ich eine load store operation machen wenn muss ich dieses Set Less Than ausführen, also muss eine Subtraktion machen. Und wenn ich eine 1,0 Anliegen habe auf diesen 2 Bits, dann will ich eine arithmetische Operation machen. Und welche arithmetische Operation? Das wähle ich dann durch die 6 Bits hier unten aus. So, hier haben wir natürlich bei Lutz da eine Addition, aber diese Addition ist eine Adressberechnung. subtraktion und eben diese logischen hands uns die wird dann durchfangt ausgewählt so wie immer wenn wir jetzt eine logik schaltung aufbauen wollen machen wir unsere funktions tabelle und die funktions tabelle sagt also die bildbelegung und das entsprechende ergebnis dafür das heißt für diese transfer befehle wollen wir die zwei bits aus dem alu opcode mit 00 belegen und die die wird mit 010 und 010 berechnet. Warum ist die ALU-Control gleich? Naja klar, ich berechne hier eine Adresse und der Adresse ist es egal, ob ich unter dieser Adresse lesen oder schreiben will. Das ist ja die reine Adressberechnung, deshalb ist es für Load und Store das gleiche. Bei Branch Equal, da muss ich natürlich entsprechend bei der ALU so eine Subtraktion ausführen. Mit diesen

sich um eine branch equal anweisung und diese bits signalisieren mir hier dass ich eine subtraktion ausführen will ja sie sehen es ja auch hier wenn ich dann den rtype befehl habe das ist die 1 0 da habe ich jetzt fünf möglichkeiten für ein rtype befehl und natürlich die subtraktion hier gleich kodiert 110 ist die funktion ja wie hier weil das hier ist ja auch eine subtraktion die addition ist stand ist mit 111 kodiert gut also ausgehen auf den informationen dieser tabelle also das ist hier unser ergebnis sind die drei bits von der alu control und alu opcode und funct das sind unsere eingabe bits können wir jetzt einfach eine logik konstruieren die diese control umsetzt ja wir bohren das ganze auf zu einer kompletten tabelle also alu op1 alu op0 und dann die 6 2, 1 und 0 und ausgehend aus dieser Tabelle leiten wir dann mit disjunktiver Normalform unsere Schaltfunktion ab, dann vereinfachen wir diese Schaltfunktion und setzen dann diese Schaltfunktion in eine Schaltung um. Das ist also vorgehensweise Grundlagenrechnerarchitektur oder digitale Systeme, müssten Sie kennen aus vorhergehenden Semestern, es ist immer wieder das gleiche. Und die Schaltfunktion nach Vereinfachungen sieht dann so aus, dass die Alu-Kontrollleitung 2 ist Alu-Op 0 oder Alu-Op 1 und F1 und so weiter. Also das sind jetzt die Schaltfunktionen in disjunktiver Normalform mehr oder weniger. Und dann setze ich das Ganze in eine Schaltung um. aus dem Alu-Obteil und diese 2 Bits und diese 6 Bits werden eben hier mit diesen Gattern miteinander verbunden und als Ergebnis kommen dann hier 3 Bits raus, nämlich meine Alu-Control-Leitungen. Gut, damit habe ich also die Alu-Control und das ist aber nur natürlich ein kleiner Teil, Das heißt, ich muss erstmal jetzt nochmal in meine Schaltung gehen und schauen, wo ist überall ein Multiplexer, was muss alles angesteuert werden durch Kontrollleitungen. Ich muss also alle Steuersignale bestimmen und dann muss ich eine Kombinatorik entwickeln, dass eben diese Steuersignale mir im richtigen Schritt, also in einem Schritt erzeugt und zwar die korrekten Steuersignale für alle Multiplexer, die da auftauchen. eingehen und wir wissen dass unser opcode immer die obersten sechs bit sind wir sagen also zum opcode op wird abgekürzt und das sind eben die bits 5 bis 0 das sind sechs bits unseres opcodes und die bestimmen sich aus den bits 31 bis 26 unserer instruktion und dann unterscheiden wir bei rtype befehlen und store befehlen und branch equal da haben wir zwei leseregister und zwar in den unsere Instruktion und 20 bis 16. Dann haben wir noch ein Basisregister im Loadstore, das ist 25 bis 21 in unserer Instruktion, beziehungsweise wenn wir mit dem Offset arbeiten, dann ist es im Bit 0 bis 15. Wir haben hier nochmal die entsprechenden Formate und wir beziehen uns eben auf diese Bitstellen, wir haben hier Bit 0, wir haben hier Bit 31 unseres Befehlsformats weil das ist ja praktisch die Bits, die wir ansapfen, um dann unsere Kontrollleitungen entsprechend zu setzen. Gut, Zielregister bei der Load-Operation steht im Befehlsfeld 20 bis 16 und beim R-Type 15 bis 11. Da haben wir ein bisschen einen Unterschied. Wir haben nämlich hier, das ist ein I-Format-Befehl, so ein Load, und da haben wir hier drin stehen das Ergebnisregister ja das ist ein bisschen unschön, da muss man umschalten, also wenn wir dieses Befehlsformat haben, dann steht das, wann es ins Registerpfeil geschrieben wird, soll an dieser Stelle die Adresse und beim Loadbefehl steht es an dieser Stelle die Adresse. Das heißt wir brauchen hier auf jeden Fall einen Multiplexer, der uns umschaltet zwischen R-Format und I-

Format Befehlen, damit das richtige Ergebnis ins Registerpfeil zurückgeschrieben wird. Okay, also jetzt bräuchten wir eigentlich zwei Projektionsmöglichkeiten, unter Umständen ein bisschen hin und her springen. Schauen wir uns erstmal jetzt die gelb hinterlegten Steuersignale an. Das sind alle Multiplexer. Wir haben 1, 2, 3, 4 Multiplexer. Und diese Multiplexer schauen wir uns einfach mal auf der nächsten Folie an. Wir haben also hier einen Multiplexer fürs Register Destination. Damit schalten wir also um zwischen den Instruktionsfiltern 15 bis 11 und 20 bis 16. Das ist der Fall, den wir gerade besprochen haben. Ich habe das Ergebnis eben bei einem Load anders als bei einem R-Type Befehl. Deshalb müssen wir hier mit Registered Destination umschalten zwischen diesen beiden Möglichkeiten. Beim ALU-Eingang muss ich natürlich auch umschalten, ob jetzt dieser Operant hier, ob der aus der Konstanten kommt, dann läuft er hier über Design Extension rein oder ob er aus dem Register-File kommt. Dann haben wir hier den Multiplexer, den wir brauchen, um zu entscheiden, ob es jetzt ein Sprungbefehl war und der genommen wird oder ein normaler Befehl. Und dann haben wir hier nochmal diesen Memory to Register Multiplexer, der entscheidet, ob das Ergebnis der ALU durchgeschaltet wird und ins Registerpfeil zurückgeschrieben wird. dieser Weg hier durchgeschaltet. Also das sind die vier Multiplexer, die ich brauche. Dann habe ich rosa hinterlegt Register Write Befehle. Die liegen, wo liegen, da haben wir Register Write, also hier haben wir ja das Register File, da muss ich sagen, wenn jetzt geschrieben werden soll, dann haben wir hier ein Memory Write und ein Memory Read. Das brauche ich natürlich für Load und Store. zugreife muss ich entsprechend hier über diese kontroll leitungen das natürlich entsprechend aktivieren so und dann brauchen wir für den branch entsprechend auch noch kontrolle nämlich ich muss über pc src umstellen die branch target address also das sprungziel ja ob ich jetzt eben oder ob ich das berechnet entnehme. Gut, das sind also jetzt alle Steuersignale. Dazu kommt natürlich noch die ALU-Control. Die ALU-Control ist die Logik, die wir ein paar Folien vorher berechnet haben. Und in diese ALU-Control geht natürlich als Eingang rein aus meiner instruktion und die alu ob als eingang und als ausgang eben die drei pizze also zwei bis über alu ob 6 bis über den fangteil und unsere alu logik alu control logik und macht dann drei pizze daraus die diese alu hier ansteuern ok so jetzt kann ich dann eine tabelle aufstellen mit allen signalen für die Befehls Typen, einmal für den R-Type Befehl, also logischer Befehl, einmal für den Load Befehl, einmal für den Store Befehl und einmal für den Branch-Equal Befehl. Hier sind also alle Kontrolle oder Steuerleitungen angegeben und hier sind dann eben entsprechend die Belegungen. Ich habe also beim R-Type habe ich natürlich Register Destination, das Ergebnis wird da reingeschrieben, das ist eine 1, Allo Source, der Multiplexer ist eine 0, Memory to Register Also Sie müssen einfach die Folie hier vorher nehmen, müssen schauen, wo diese Steuersignale anliegen und dann eben sich nochmal vergegenwärtigen, warum da jetzt eine 0 steht, warum da eine 1 steht, um das Ganze genau zu verstehen. Und es gibt natürlich auch so ein paar Don't Cares. Die Don't Cares sind wie immer mit X gekennzeichnet. viele die eingabesignale also register destination allo source memory to register register write also eben die entsprechenden ansteuerungen also im register destination ist natürlich eins weil ich aus meiner

instruktion sehe welches destination register ich beschreibe ja da steht ja in der Wenn ich einen R-Type Befehl vorliegen habe, dann möchte ich natürlich, dass als zweiter Operand an der ALU das anliegt, was im Registerpfeil steht, also das ReadData2 ist der ALU-Input, das wird damit gesteuert und so weiter. für diesen Befehlstyp R-Type Befehl den Wert 0 oder den Wert 1 hat. Gut, wir haben hier noch ein weiteres Beispiel, oder das Beispiel einfach noch ein bisschen ausgebaut für diese Eingabesignale, wo Sie wirklich genau nochmal dann sehen, aus der Instruktion, was ich da rausziehe. 32 Bits praktisch aufgeteilt, wo steht das RS Register, wo steht das RT Register, R-Type Befehle haben dieses Format, wir haben Opcode 6 Bits, RS 5 Bits, RT 5 Bit, RT 5 Bit, Shift Amount 5 Bit und Funct 6 Bit und hier steht nochmal, der RS Teil von hier, der geht hier ein, der RT Teil, der geht hier ein, der Destination Register Teil geht hier ein, Der Opcode wird hier verarbeitet. Also das soll Ihnen einfach widerspiegeln, dass tatsächlich ein Bitmuster in meinem Maschinenbefehl nichts anderes ist als eine Ansteuerung meiner Hardware. So kommen Sie also direkt aus dem wilden Bitmuster in dem Maschinenbefehl dazu, dass Ihre Hardware direkt angesteuert wird. Wir geben stellenweise unseren Opcode natürlich Dezimal an, obwohl der natürlich binär kodiert ist, das ist klar, einfach abkürzende Schreibweise. Und wenn alle Bits des Opcodes auf 0 sind, dann weiß ich, ich habe einen R-Zeitbefehl. also 1 0 0 der steht für einen branch equal befehl das ist die zuordnung die ich dann halt irgendwann mal treffen muss ja das ist einfach ich hinterlege praktisch eine tabelle mit der entsprechenden zuordnung für alle befehle das sind jetzt hier die überbegriffe aber wir wissen ja es gibt eben unterschiedliche entschuldigung es gibt unterschiedliche loadbefehle dann unterscheiden die sich halt noch mal im detail aber das ist jetzt so mal die obermenge der belegung für den So die Main Control die müssen wir natürlich noch bauen, die haben wir ja bis jetzt immer so ja auf magische Weise macht die ihre Aufgabe richtig und dafür müssen wir erstmal eine Ausgabe betrachten. In die Main Control ein gehen nur der Opcode, das heißt die obersten 6 Bits meiner Instruktion, entspricht dem opcode 5 bis 0 also der bits 5 bis 0 sind also sechs bits die in die main control eingehen und die main control berechnet daraus die belegung für diese steuerleitungen alu op das ist eine 2 bit leitung die hier belegt wird und ansonsten sind es ein bit leitungen die hier teil der gewissermaßen auch zur operation gehört aber den brauchen wir ja jetzt nur für die alu control und deshalb brauchen wir den nicht in die main control mit betrachten ja für die main control sind nur diese zwei bits der alu op entscheidend die wir auch brauchen für die alu steuerung aber darüber hinaus brauchen wir noch die bits aus dem funkt teil und da haben wir eine extra logik für die alu control gebaut da gehen diese zwei bits als Wir bauen uns also jetzt wieder so eine einfache Steuerung, indem wir die Befehle betrachten. Wir haben also diese sechs Bits, die wir auswerten müssen und wir haben eine Belegung festgelegt für den R-Teil, Loadword und Storeword und Branch-Equal-Teil. Und ja, jetzt ist einfach wieder ablesen aus dieser Tabelle und dann in die Schaltung schauen, dann wissen wir genau, wann diese Steuerleitungen aktiviert werden müssen. den r-teil befehl load word befehl store word befehl branch equal befehl das sind unsere eingänge hier haben wir ein bootcutter entsprechend anliegen und das gibt mir dann direkt als ausgang

meine steuerleitungen und die muss ich dann halt an die richtigen ausgänge meiner schaltung Kennen Sie aus vorhergehenden Vorlesungen, wie ich so eine Logik auswähle. Wir haben hier unsere Eingänge OP0 bis OP5 und wir haben hier unsere Ausgänge, das sind die einzelnen Steuerleitungen und das ist die Verknüpfung, die sich ergibt aus dieser Tabelle hier. Okay, jetzt können wir also unseren Single Cycle Datenpfad inklusive der Main Control einzeichnen. kommen diese main control speist sich als eingang aus unserem instruktionsregister über den sechs bits und diese main control steuert also alle multiplexer die in unserer schaltung auftauchen und diese main control gibt mir auch zwei bits hier aus für die alu operation die gehen hier unten auf die alu control und außerdem geht der funktionsteil also aus dem instruktionsbit von berechnet die Steuersignale für die ALU und steuert hier, dass immer die richtige Operation bei der ALU ausgeführt wird. Gut, das ist also jetzt der komplette Single-Cycle-Datenpfad. Da ist jetzt alles dabei, alle Steuerleitungen, alle Register, alle kombinatorischen, alle speichernden Elemente. Und was wir jetzt noch machen müssen, wir müssen unseren Takt dimensionieren. Wir müssen schauen, dass für einen Befehl, der läuft ja von hier, wo ich den Befehlsteller auswerte, bis zum Ende praktisch, bis alles berechnet ist und das Ergebnis ins Registerpfeil zurückgeschrieben ist. Dafür müssen wir jetzt die Dauer ausrechnen und zwar für den Befehl, für den es am längsten dauert. Und dann müssen wir einfach schauen, wie lange es dauert, um die richtige Taktzeit angeben zu können. ist gleichzeitig dass natürlich diese tag periode die ich wähle durch den längsten pfad bestimmt wird und wenn man ein bisschen genauer drauf schaut dann sieht man dass das laut der nächsten pfad darstellt da wir einen single cycle tatenpfad haben ist klar dass unser cycles by instruction ist immer eins ja wir haben unseren takt so dimensioniert dass ich den längsten befehl abarbeiten kann alle anderen befehle sind ja eher kürzer aber es wird nicht kürzer getaktet also und es ist natürlich auch klar, wenn ich mich nach dem langsamsten richte, dass ich dann noch ein bisschen Potenzial in der Performance nach oben habe und das ich auch nutzen möchte. Und je größer der Unterschied zwischen dem schnellsten und dem langsamsten Befehl ist, desto mehr Potenzial habe ich natürlich. Das heißt, der Leistungsvorteil, wenn ich dann mal auf Multicycle gehe, der erhöht sich, wenn ich entsprechend große Unterschiede habe. bei unserer Single-Cycle-Datenfahrt ist, da kann ich kein Pipelining machen. Ja, weil jede Einheit, die ich da habe, die wird maximal einmal pro Befehl genutzt und damit ist kein Pipelining möglich. Ich habe da also keine Parallelität drin. Das muss ich erst irgendwie ermöglichen. Und das sehen wir dann über den Umweg des Multi-Cycle-Datenfahrts, wie ich so ein Datenfahrt-Pipeline-Fähig überhaupt mache. Fall mit so ein paar Annahmen. Also beim R-Type Befehl, alle Befehlstypen müssen auf das Instruktionsregister zugreifen, klar ich muss mir den Befehl holen. Dann alle lesen Register bis auf den Jump Befehl, weil der nur mit dem Immediate funktioniert. Alle verwenden die ALU, die Laufzeiten nehmen wir hier immer gleich an. Die Load-Store Befehle greifen auf den Speicher zu sehen deshalb das hier ist der längste pfad ja 10 plus 5 plus 10 plus 10 plus 5 ist 40 ist also der längste pfad und gleichzeitig haben wir ein benchmarking gemacht um so die häufigkeit dieser befehle zu bestimmen und da sehen wir dass also fast 50 prozent sind rtl befehle 22 prozent

sind die mittel load befehle 11 prozent stopp befehle 16 prozent branches und 2 prozent jumps die wir da haben eine relative Häufigkeit, äh Quatsch, eine mittlere Ausführungszeit berechnen, indem wir eben diese relative Häufigkeit, die wir haben, zugrunde legen, als Gewichtung praktisch und dann kommen wir darauf, dass wir im Mittel 31,6 Nanosekunden bräuchten, wenn wir jeden Befehl maßgeschneidert den Takt machen würden und das Potenzial, das ich also habe, wenn ich mich nach dem langsamsten richte, ja hier haben wir 40, haben wir 40 zu 1 zu 31,6, 1,27 mal schneller sein wenn ich meine takte maß schneidet also es ist schon jetzt nicht ganz groß berauschend aber es steckt doch ein bisschen potenzial drin so dass wir uns also durchaus ebenso ein multi cycle datenpfad mal anschauen können ok ich denke an dieser stelle zum Multicycle-Datenpfad und wenn ich das jetzt beginne, dann muss ich das nächste Mal nur wiederholen. Wir werden also in der nächsten Vorlesung uns eben darüber unterhalten, wie wir unseren Datenpfad ändern müssen, damit er Multicycle-fähig ist und am Ende der Vorlesung dann das Ganze als Pipeline betrachten. Ich hoffe, Sie haben ein bisschen ein Feeling dafür bekommen, dass einerseits der Datenpfad nicht so ganz kompliziert ist, wie man sich das vorstellt, doch im Detail wieder der Teufel ein bisschen steckt, dass man also wirklich alle Steuerleitungen, alles aufbauen, alles betrachten muss. Versuchen Sie das einfach alles zu verstehen, wie das zusammenarbeitet. Das hilft Ihnen dann auch in der nächsten Vorlesung den Multi-Cycle-Datenpfad besser zu verstehen.