Guten Morgen meine Damen und Herren, auch heute wieder zu sehr früher Stunde beschäftigen wir uns mit der Vorlesung Rechnerstrukturen. Single Cycle Datenpfad. Was war die Idee? Die Idee war, dass wir einen kompletten Maschinenbefehl innerhalb eines Taktzyklus abarbeiten. Dadurch mussten wir uns natürlich nach dem langsamsten Befehl richten, das heißt der Takt muss ja so dimensioniert werden, dass dann eben alle Befehle erledigt werden können innerhalb eines Taktes, das heißt man schaut auf den langsamsten Befehl Gut, das hat natürlich Nachteile, weil potenziell kürzere Befehle natürlich dann genauso lang laufen. Und deshalb legt die Idee nahe, dass man sich mal Gedanken macht über einen Multicycle-Datenpfad. Und das ist das, was wir in der heutigen Vorlesung machen wollen. Und die Grundidee ist die, dass ich erstmal diesen Befehlszyklus versuche aufzuteilen in ein paar kleinere Zyklen durchlaufen muss und wenn nein eben diese entsprechenden zyklen einsparen so wir gehen jetzt mal aus von unseren single cycle datenfahrt und machen ein paar änderungen damit wir dann zu einem multi cycle datenfahrt kommen können erstmal wir lösen diese harvard architektur wieder auf das heißt wir die Instruktionen hier und einen Speicher für die Daten hier. Und aus diesen beiden Speichern machen wir wieder einen Speicher. Das heißt, wir haben einen Speicher für Daten und Befehle, auf den wir dann zugreifen wollen. Da wir ja unsere Instruktion während der ganzen Zyklen immer wieder benötigen, müssen wir unsere Instruktion zwischenpuffern. ins instruktionsregister und steht hier eben für alle zyklen dann zur verfügung gut außerdem haben wir noch eine kleine änderung bei den alus wir wollen nicht mehrere alus haben sondern wir wollen eine alu haben und dementsprechend mit der wir alles berechnen also auch die adressberechnung machen auch die program counter erhöhung machen und dementsprechend sparen wir diese beiden die mehrfach nutzbar ist und die diese Berechnungen jeweils ausführt. Was noch nicht eingezeichnet ist, wir werden noch ein paar Multiplexer brauchen, inklusive Steuereingänge, um eben Daten- und Kontrollflüsse ein bisschen zu steuern, weil wir ja die Hardware noch ein bisschen verändern. Die ALU ändert sich nicht großartig, also wir haben ja eine ALU, die 5 Operationen kann und die ALU Operation Control, also die bleibt gleich. beantworte, also sowohl arithmetische Befehle, also R-Type Befehle, als auch eben Branches und Jumps, also I- und J-Type Befehle und die manipulieren natürlich den Program Counter und da müssen wir auch aufpassen, dass das richtig umgesetzt wird im Multicycle Datenpfad. Okay, das ist jetzt hier eingezeichnet, was eben noch dazu kommt an Änderungen. Wir brauchen PCWrite dafür wann wird der Programmkader geschrieben Memread, Memwrite brauchen wir, wir haben ja nur noch einen Speicher da stehen Instruktionen und Daten drin und ich muss unterscheiden können ob ich jetzt von diesem Speicher lese oder schreibe dann haben wir unser Instruktionsregister auch hier müssen wir steuern wenn da reingeschrieben werden soll mit dem Instruktionsregister write und also da steht immer dann unser Maschinenbefehl gepuffert drin in unserem Instruktionsregister und dann haben wir hier unser Register File Register File muss natürlich auch angezeigt werden, Register Right, wann an diesem Eingang Daten übernommen werden sollen. Das sind also diese fünf Änderungen da oben. Und dann haben wir zusätzlich eben Multiplexer. Der Multiplexer 1 bezieht sich hier auf dem Program Counter. Da

wird praktisch unterschieden, ob der reguläre Befehl ausgeführt wird oder ein Sprungbefehl ausgeführt wird. Also das ist hier nochmal erklärt. Und zwar kann entweder das ALU-Result anliegen oder der Programm-Counter anliegen und gesteuert wird es über diesen Multiplexer 1, was praktisch an Signal jetzt hier als Adresse in den Speicher gegeben wird. Dann haben wir noch einen ALU-Input-Selektor. und hier kann über Multiplexer gesteuert werden, ob jetzt der Input vom Program Counter kommt oder ob der Input aus Register File kommt. Und hier haben wir eben auch vier verschiedene Möglichkeiten, wo der Input herkommen kann. Er kann von der Immediate Konstante kommen, die dann auf 32 Bit aufgefüllt wird und zur Not geschifftet wird. Und wir können hier die Konstante 4 durchschalten. Das brauchen wir, wenn wir regulär den Befehlsteller um 4 erhöhen. Und hier haben wir dann auch nochmal eine Eingangsleitung, die wir zurückverfolgen können. Das kommt aus dem Instruktionsregister, das ist auch so eine aufgefüllte Immediate Konstante, die da anliegen kann. So, und dann haben wir noch den RegisterWriteDataSelector. Das ist eben hier unten dieser Multiplexer. Eingang und das kann eben von hierher kommen, das heißt direkt aus dem Speicher. Das brauche ich zum Beispiel, wenn ich so eine Load-Operation mache und mir Daten aus dem Speicher hole und direkt ins Register schreibe, dann brauche ich diesen Eingang und wenn ich das ALU-Ergebnis zurückschreiben will, ins Register feile, dann brauche ich diesen Eingang und gesteuert wird es eben über diesen Multiplexer 4. schon haben wir die wichtigsten Vorbereitungen für unseren Multi-Cycle-Datenpfad getroffen. Es gibt noch so ein paar Hinweise auf diese Folie, man könnte das natürlich auch anders bauen, also vielleicht ist das ein bisschen schwer immer gleich zu durchschauen, diese ganzen Multiplexer und Selektoren, also anstelle von so einem Allo-Input-Selektor könnte man natürlich auch so eine 3-Bus-Struktur aufbauen, das heißt ich verbinde direkt Quelle und alle auf diesen Bus zugreifen, aber da muss ich natürlich ein Protokoll fahren, das sichergestellt ist, dass eben nur ein Sender pro Bus aktiv ist. Also das ist eine Schreibweise, also eine Darstellungsweise, die ein bisschen anders ist, aber das Ziel ist natürlich das gleiche. Ich will eine Leitung durchschalten und das kann ich jetzt mit so einem Selektor machen, mit einem Multiplexer oder ich kann es mit einer Busstruktur machen. multi cycle datenfahrt dann brauche ich eben bei zusätzliche register weil ich eben mit mehreren zu den mehreren zyklen unterwegs bin und deshalb muss ich zum beispiel des instruktionen die ich mir aus dem speicher hole die muss ich zwischenspeichern weil eben die gebraucht wird für unterschiedliche zyklen und dementsprechend kann ich dann natürlich nicht jedes mal auf dem und muss eben diese Instruktion mir puffern. Jetzt wollen wir uns dann mal unseren bekannten Befehlszyklus anschauen und wollen uns überlegen was da genau passiert und es liegt natürlich hier auch ein bisschen nahe dass wir, wir haben ja schon so eine Aufteilung in fünf Phasen dass wir diese fünf Phasen eben getrennt betrachten und das praktisch dann auch Also wir haben die erste Phase, das ist die Instruction Fetch Phase und diese Phase ist natürlich für alle Befehlstypen gleich. Wir unterscheiden ja immer drei Befehlstypen, das ist die R-Type Befehle, das sind also diese arithmetischen Befehle, dann haben wir Memory Reference Instructions, also das sind Load-Store Befehle und dann haben wir eben Branches, also Sprünge und bedingte Sprünge, das ist die dritte

Klasse von Befehlen, die wir hier betrachten wollen. Und die Befehlsholphase ist natürlich für alle gleich. Was mache ich? Ich lade mein Instruktionsregister, das ich neu eingeführt habe, aus dem Speicher und die Instruktion, die ich brauche, die finde ich unter der Adresse Program Counter. Anschließend deute ich gleich auf den nächsten Befehl, das heißt mein Program Counter wird um 4 erhöht. Also das ist für alle drei Befehlstypen ganz genau das gleiche. Genauso die zweite Phase, die Instruction Decode und Register Fetch Phase. Hier bestimme ich ja praktisch meine Operanten und der eine Operant ist eben der Operant A und den adressiere ich über das Instruktionsregister und zwar die Bittstellen 25 bis 21 und der andere Source-Operant, der steht in den Bittstellen 20 bis 16 und das sind also meine zwei Eingangsoperanten für arithmetische Operationen natürlich das sprungziel und das sprungziel ist der aktuelle program counter und dann kommt diese immediate noch oben drauf die eben entsprechend vorbereitet wird dass es die richtige adresse ergibt das heißt die muss ich um zwei schriften also mit vier multiplizieren byte und wort adressierter speicher ist da das stichwort dazu und ansonsten haben wir unsere sign extension und die immediate wird einfach das Sprungziel berechnet. Okay, das ist also Instruction Decode für die drei Befehlstypen. Und die Execute-Phase, die ist natürlich jetzt ein bisschen unterschiedlich. Also wenn ich einen arithmetischen Befehl habe, da wird die Operation ausgeführt. Das Ergebnis kommt nach ALU-Output. Wenn ich hier eine Memory Reference Instruction habe, also ein Load Store, dann wird hier die Adresse berechnet, unter der ich dann auf den Speicher zugreifen muss. output ergibt sich eben, das ist eine Adressrechnung, also der eine Operand für die Adressrechnung ist hier, ist der Operand A und der andere berechnet sich aus der immediate, die in dem Befehlswort drin steht, die eben entsprechend dazu addiert wird. Und wenn ich einen Branch habe, also einen bedeckten Sprung, dann wird ausgewertet, ob A gleich gleich B ist, dann kommt eben 0 raus, dann wird also das 0-Bit gesetzt und wenn das 0-Bit gesetzt wird, dann wird hier einfach der Program Counter umgeladen und zwar bestimmt sich dann der neue Program Counter aus der Branch Target Adresse. Die Branch Target Adresse habe ich für den Fall des Sprungbefehls hier schon vorbereitet, vorberechnet und hier wird dann der Program Counter umgesetzt auf die Branch Target Adresse. Gut, also das heißt das Execute ist natürlich für diese drei Befehlstypen ein bisschen unterschiedlich dann habe ich als nächste aktion habe ich eben einen speicher zugriff oder ich berechne praktisch das ergebnis zu ende für rtype instructions also wenn ich eine rtype instruktion habe dann schreibe instruktionsregister also die bit 15 bis 11 sagen mir welches register ich im registerfall beschreiben soll und ich hole mir praktisch den alu output ins register file und bin damit für rtype operationen fertig das heißt da wird in dem sinn gar kein writeback mehr ausgeführt da brauche ich ja nicht mehr weil ich berechne das ergebnis und ich habe eine register registermaschine schreibt das ergebnis ins register file und damit ist die reference maschinen befehl habe dann muss ich bei einem load natürlich eben aus dem speicher lesen und beim store muss ich eben aus dem speicher schreiben so das heißt beim load sind die die speicherdaten ergeben sich eben aus der adresse alu output und damit greife ich auf den speicher er meine Adresse berechnet, also alle Output berechnet, gibt die dem Speicher

mit und da speichere ich das B ab. So, damit ist die Store-Operation fertig, das Load ist aber noch nicht fertig, ich muss ja das, was ich aus dem Speicher lese, muss ich noch ins Register-File schreiben, das heißt im Register-File, und das ist jetzt ein bisschen anders als bei den R-Type-Befehlen, da ist jetzt die Adresse im Register-File in den Bits 20 bis 16, ja später natürlich auch eine Logik dafür bauen, die das umsetzt, dass ich hier unterschiedliche Zieladressen im Registerpfeil habe. Also bei der R-Instruction sind es die Bits 15 bis 11 und bei der Load-Instruction sind es die Bits 20 bis 16 und das ist also die Registeradresse in meinem Registerpfeil und beim Load schreibe ich da die Daten hin. So, und was wir schon ganz gut sehen können ist, dass RTAB-Instructions, da brauche ich eigentlich vier Zyklen, also Und bei Store brauche ich auch 4 Zyklen, bei Load brauche ich 5 praktisch und bei diesen Sprüngen brauche ich eigentlich bloß die ersten 3, weil hier nichts mehr passiert. Und das ist schon die Grundlage unseres Multi-Cycle-Datenfahrtes, weil ich dann eben entsprechend für die RTAB Instructions nur 4 Zyklen brauche, für die Load Instructions 5 und für die Sprung Instructions 3. Und das nutze ich natürlich aus, um Zeit einzusparen. Bis wir soweit sind, müssen wir noch so ein paar Kleinigkeiten einbauen. Wir müssen also unseren Multi-Cycle-Datenfaden noch ein bisschen erweitern, damit wir eben diese Sprung- und Branch-Logic mit aufbauen. Wenn ich auf den Program-Counter schaue, dann habe ich drei Möglichkeiten, den nächsten Befehl zu holen. Wenn ich ganz normale Instruktionen habe, also arithmetische Instruktionen, R-Type und Load-Store-Operationen, Befehle zur Ausführung ansteht, wird ganz normal durch die Erhöhung des Programme-Counters um 4 berechnet. Bei weitadressierter Speicher, 32-Bit-Maschine, 4 mal 8 ist 32. Also es ist immer wieder das Gleiche, wird um 4 erhöht. Wenn ich einen Branch habe, also einen E-Type-Befehl, dann habe ich ja vorher meine Branch-Target-Adresse berechnet, also meine Sprungziel-Adresse berechnet und das Sprungziel-Register, also dieses und diese neue befehlsadresse die wir dann genommen wenn die bedingung true ist also im falle eines genommenen sprungs so und wenn ich jetzt eine jtype habe also einen jump dann ist es einfach dann berechne ich einfach meine sprungadresse da habe ich ja jetzt keine bedingung und die sprungadresse ist in dem ich dann konkateniere praktisch den program counter also die obersten Bitstellen, damit ich richtig springe. Also wenn das eine 1 ist, dann ist es ja negativ, dann springe ich nach hinten. Und wenn das eine 0 ist, dann ist es positiv, dann springe ich nach vorne. Und die Sprungweite ergibt sich praktisch aus dem Maschinenbefehl. In den Bits 25 bis 0 steht die drin. Und wieder jetzt Byte und Wort adressiert, deshalb muss die um 2 geschifftet werden, damit das alles passt. eine Sprungzielberechnung in J-Type und es geht relativ schnell. So, und wenn wir uns das jetzt hier im Bild anschauen, dann haben wir deshalb diesen Multiplexer hier aufgebaut, weil der Multiplexer, der geht, wenn man den verfolgt, geht hier vorne rein in den Program Counter und der Program Counter kann sich eben aus diesen drei Möglichkeiten speisen, je nachdem, ob ein R-Type, ein E-Type, also ein Branch oder ein J-Type, also ein Jump vorliegt. und der läuft dann auf dem pc auf so hier ist noch mal erklärt für den chat befehl warum diese sprungadresse sich so berechnet ich habe ja im instruktions fort eine 26 bit konstante und die wird kombiniert mit

den vier most significant bits des program counters also damit ich eben weiß ob ich vorwärts oder rückwärts springe und dann konkateniere ich das also das sind 30 bits und schiebt es dann So mache ich dann also aus den vorher 26 Bits praktisch durch die Konkatenation und durch den Shift mache ich dann die 32 Bit Adresse, die ich brauche für die nächste Instruktion. So jetzt muss ich dann eben noch bestimmen, ob der Program Counter, wann der geschrieben wird. Und dafür brauche ich eine kleine Logik. Und diese kleine Logik, die finden Sie hier. conditional sprung also für den jump das ist ein pc ride für einen bedingten sprung ja und das ganze wird hier verundet und verodert und gibt dann das steuersignal damit eben im program counter register praktisch dieser eingang hier übernommen wird und die schaltung ist hier praktisch dass das pc true und der befehlszähler muss überschrieben werden oder ich habe das ist hier die bedingung pc write conditional also wenn ich eine bedingung vorliegen habe und die bedingung ist erfüllt ja wir haben ja diese zero bit diese zero bit zeigt an die bedingung ist erfüllt und wenn ich das dann verunde dann muss natürlich dann spring ich bedingt dann wird also der befehlszähler überschrieben oder wenn ich eben einen unbedingten sprung habe also Und diese kleine logische Formel, die finden Sie hier mit diesen zwei Gattern. Das ist das Untergatter und das steuert das PC-Write-Signal. Okay, das heißt, Sie sehen schon, die Main-Control ist im Wesentlichen gleich zum Single-Cycle-Datenpfad. Es ist ein bisschen was dazugekommen für die Sprunglogik. dazu gekommen und so langsam baut sich eben dann unser kompletter Multicycle-Datenpfad auf. Also unsere Main Control Unit kennen wir aus dem Single Cycle Datenpfad, die wird im Wesentlichen gesteuert über den Opcode im Maschinenbefehl und zwar eben die Bits 5 bis 0 des Opcodes und die Taktzyklus natürlich. Wir haben in der Vorlesung Grundlagenrechnerarchitektur schon die Control Unit ausführlich besprochen und da eben auch gezeigt, wie ich so eine Control Unit in Gattern praktisch aufbauen kann und zwar ist eben das entsprechende Kapitel Endliche Automaten, wo wir das dann behandelt haben. Wem noch Details dazu interessieren, alle Beispiele, in der Bibel des Rechenarchitekten, also Hennessy Patterson, Computer Organization and Design, der Hardware Software Interface und das ist also ein Buch, das gibt es dann schon in der 10. Ausgabe oder so, weil das einfach so ein grundlegendes Werk zur Rechenarchitektur ist, deshalb nicht erschrecken, dass es schon 1994 erschienen ist, es ist immer noch brandaktuell und da eben in dieser Ausgabe, in den neueren Ausgaben, da muss man dann halt ein bisschen schauen, wird ein bisschen weiter hinten sein, weil das Book jedes Mal ein bisschen dicker wird, wenn eine neue Auflage kommt. Okay, also, wenn wir jetzt auf unseren Multi-Cycle-Datenpfad schauen, dann haben wir festgestellt, okay, das Load braucht am längsten, weil wirklich alle einzelnen Teilphasen durchlaufen werden. Und dann machen wir jetzt wieder Simulationen und Benchmarks und schauen, Last, an der gesamten Last und dann stellt sich dabei raus, okay, das Load hat so eine relative Häufigkeit von 22%. Also 22% aller Befehle sind irgendwo Load-Befehle. Deutlich weniger sind die Store-Befehle, das ist nur halb so viel, das sind 11% und da beim Store-Befehl dieser letzte Teilzyklus wegfällt, ich muss ja nur auf den Speicher zugreifen und brauche dann nicht noch mein Register-File schreiben, habe ich einen Zyklus und 11%

ist meine relative Häufigkeit von den Stores. Die normalen arithmetischen Befehle, die R-Type Befehle, das ist die größte Anzahl, das ist also eine relative Häufigkeit von 49% und auch hier brauche ich eben 4 Zyklen, weil ich habe ja in der X-Phase berechne ich mein Ergebnis und dieses Ergebnis muss zurück ins Registerpfeil geschrieben werden und dementsprechend 4 Zyklen. und die schnellsten das ist die branch und jumps das sind eben entsprechend drei zyklen wir haben 16 prozent branches und 2 prozent jumps also das ist das wenigste ich gehe jetzt mal ein paar folien zurück damit sie sich einfach noch mal dieses cycles per instruction wo kommen diese zahlen her diese zahlen kommen im wesentlichen aus dieser tabelle hier ja sie sehen also rtype da fehlt das Beim Store fehlt dieser Teil, deshalb braucht der Store noch vier Zyklen. Der Load hat 1, 2, 3, 4, 5 Schritte, muss alle durchlaufen, da wird in jedem Schritt was passieren. Und bei Branch & Jump, da haben wir nur diese drei Phasen und hier passiert nichts. Also dementsprechend speist sich diese Tabelle hier aus den Festlegungen in der Tabelle vorher, welchen Befehlstyp zu tun. Das heißt, es ist relativ realistisch, was hier steht und jetzt können wir natürlich unsere Leistung berechnen und die Leistung ist auch wieder einfach, also das mittlere CPI, was ich brauche, das ist die Summe über alle CPIs gewichtet mit relativer Häufigkeit und dann komme ich auf einen Wert von 4,04. hätte dann müsste ich ja jeden befehl in einem zyklus erledigen und ich müsste meinen zyklus so groß dimensionieren wie der langsamste befehl ist das heißt wie der load befehl ist das heißt ich bräuchte immer 5 cycles per instruction für jeden befehl und dementsprechend ist der geschwindigkeitsgewinn den ich erziele ist der faktor 1,24 ich setze einfach in relation des cycles per instruction für des Cycles per Instruction für einen Multicycle-Datenpfad und ja, dividiere ich es auseinander, also setze ich es in Relation und habe damit dann die Leistungsverbesserung. Okay, also, es würde sich natürlich lohnen, das lernen wir daraus, wir bräuchten dann natürlich eine Steuerung, die erkennt, um welchen Befehlstyp handelt es sich und dementsprechend viele Zyklen dann zu reservieren für die Abarbeitung dieses Befehls, das ist eine einfache Noch viel interessanter ist natürlich, wenn wir die Erfahrungen, die wir jetzt gesammelt haben, einfach nutzen, um uns zu bauen aus den Erfahrungen des Single- und Multicycle-Datenpfads einen Pipeline-Datenpfad. Weil erst wenn ich mehrere Befehle gleichzeitig bearbeite, dann habe ich einen echten Geschwindigkeitsgewinn. Also nochmal einen viel deutlicheren Geschwindigkeitsgewinn. Beim Pipelining ist immer klar, es ist egal ob bei VW in der Fabrik das ist oder ob ich das jetzt in der Rechnerarchitektur mache, erstmal muss ich meine Instruktion in einzelne Schritte aufteilen. Fetch, Instruction, Decode, Execute, Mem und Write Back sind. Wir wollen aber jetzt ausgehen nicht vom Multicycle-Datenpfad, sondern wir wollen ausgehen vom Single-Cycle-Datenpfad. Und zwar wollen wir den eben so umformen, dass die Hardware-Einheiten jeweils einem Schritt genau zugewiesen werden können. Der Vorteil des Single-Cycle-Datenpfads für diese Vorgehensweise ist, weil die Schritte einfach besser abgetrennt sind, eben Harvard-Architektur habe, also einen Instruction-Speicher und einen Data-Speicher und das ist an der Stelle für einen Pipeline-Datenpfad ein bisschen hilfreicher. Was zusätzlich dazu kommt, ist Zwischenergebnisse speichern. Ich brauche irgendwelche Pipeline-

Register, weil ich zwischen den Stufen der Pipelines Ergebnisse vorliegen habe, die ich noch brauche. Die muss ich also in Pipeline-Register zwischenspeichern. aufbauen und zwar für unsere charakteristischen Befehle, die wir jetzt die ganze Zeit verwendet haben, also für einen arithmetischen Befehl, für einen Load-Store-Befehl und für Branches und Jumps. Dann wollen wir uns schauen, welche Steuersignale brauche ich, wie müssen die generiert werden, dass ich eben mehrere Befehle gleichzeitig bearbeiten kann. Ich muss meine Main-Control-Unit redesignen und Parallelverarbeitung führt zwangsläufig immer zu Konflikten durchsprünge und diese konflikte die muss ich einerseits erkennen ja und dann wenn ich sie erkannt habe dann muss ich diese konflikte irgendwie lösen und das am besten natürlich voll automatisch von der hardware ohne dass der user da als großen einfluss drauf hat ok also schauen wir uns unser singles das ist jetzt noch mal der single cycle datenpfad wie wir ihn ist jetzt naheliegend ich habe eine instruction fetch phase da wird die instruktion eben im wesentlichen nur aus dem instruktionsspeicher gelesen das ist es und der befehlszähler um vier erhöht ja das ist die normale instruction fetch phase hier ist der befehlszähler der wird um vier erhöht wird dann zurückgeschrieben und hier ist mein instruktionsspeicher und instruktion wird ins register geladen das passiert in instruction fetch instruction decode stellt mir Die Execute Phase, da wird eine ALU Berechnung durchgeführt. In der Memory Access Phase, da wird eben für LoadStore auf den Speicher zugegriffen, für LoadLesen, für StoreSchreiben. Und bei der WriteBack Phase, da wird das Ergebnis dann ins Registerfile zurückgeschrieben. Also wenn ich auf dem Speicher lese, beim Load zum Beispiel, wird es hier zurückgeschrieben. Ich kann, wenn ich jetzt nur eine Berechnung mache, diesen Speicher natürlich also für eine arithmetische Operation zum Beispiel umgehen kann über diesen multiplexer direkt mein ergebnis ins registerfeil zurückschreiben diesen speicherteil den brauche ich jetzt nur wenn ich lotster operationen habe gut dann macht man einfach so ein bisschen hardware betrachtungen und versucht also halbwegs vernünftige annahmen zu treffen aufgrund der anzahl der kratter die da durchlaufen werden müssen und wir haben jetzt einfach mal eine annahme gemacht für die zeitanforderungen also nehmen eher so als relative Zahl. In der Instruction-Fetch-Phase, da habe ich einen Zugriff auf Instruction-Memory und dann ist noch ein bisschen Logik dabei, also sage ich, da brauche ich so 10 Zeiteinheiten, in dem Fall Nanosekunden als Beispiel. Instruction-Decode ist ja ganz einfach, das ist ja nur das zur Verfügung stellen von Operanten, das schaffe ich in 5 Zeiteinheiten. Die Execute-Phase ist die ALU, die durchlaufen wird, sagen wir mal, da brauchen wir 10 Zeiteinheiten. Der memory dann mit zehn einheiten dimensioniert und so ein register schreiben das ist ähnlich wie hier in der id-phase da wird ein register gelesen hier wird eins geschrieben register ist ja immer schneller als speicher also nehmen wir hier fünf nanosekunden aus jetzt wollen wir ja mehrere befehle gleichzeitig bearbeiten und müssen unsere pipeline irgendwie takten und es ist klar dass Hier drei langsame Phasen, zwei schnelle Phasen und eben die langsamste Phasen sind 10 Nanosekunden. So, jetzt ist es so, wir treffen ein paar Annahmen, damit das Ganze nicht zu kompliziert wird. Also realistische Pipelines, die werden noch viel komplexer. Also eine Annahme ist zum Beispiel, dass wir jetzt keine Multicycles

Befehle betrachten. befehl aber also floating point division dann kann es sein dass die phase x eben nicht in einem zyklus erledigt wird sondern dass ich für die phase x zum beispiel drei zyklen braucht also statt zehn nanosekunden für die phase x 30 nanosekunden brauche so was ist in dritten der realität natürlich auf aber wir wollen es jetzt nicht betrachten das heißt wir gehen davon aus dass jeder befehl genau über einen eine zeiteinheit für diese zyklen entsprechend braucht sekunden das wir da alles erledigen können gut was wir jetzt machen müssen eben die register die wir brauchen und haben müssen wir zu den befehls phasen zuordnen und am besten drücken wir das aus durch register transfer operationen das heißt was passiert in den einzelnen phasen wer ist beteiligt und was muss steht hier nochmal was passiert ist klar instruction fetch zugriff auf den speicher instruktionsspeicher befehls zähler brauchen wir und wir brauchen eine alu um eine befehls zähler zu implementieren instruction decode wir müssen unsere register lesen wir müssen unter umständen wenn wir mit einer immediate also mit einer konstanten arbeiten müssen wir mit der sign extension unit die auf 32 bit auffüllen weil unsere konstanten eben normalerweise im instruktionswort wir die ALU und dann eben bei Speicherzugriffsoperationen machen wir hier eine Adressberechnung. Dann in der MEM-Phase Zugriff auf den Speicher und Writeback schreiben wir auf die Register. Wenn wir uns jetzt anschauen auf der nächsten Folie, ist die Datenflussrichtung immer von links nach rechts für unsere Pipeline und es ist natürlich gut, wenn sich alles in eine Richtung immer bewegt, dann kann ich das hintereinander schön einphasen. die rückwärts gehen nämlich das writeback schreibt natürlich ins register file das heißt da geht es dann nicht von links nach rechts sondern von rechts nach links und wenn ich natürlich einen sprung mache und die neue die sprung adresse berechne dann geht ihr auch zum programm counter zurück und das problem ist wenn ich rückwärts gehe dass nämlich diese ausnahmen den nächsten befehl betreffen und nicht den aktuellen befehl das heißt das haben wir hier unser bild Und es läuft eigentlich, die ganzen Daten laufen schön durch unsere Architektur durch. Links beginnen, nach rechts bis das Ergebnis vorliegt. Und das sind die zwei Pfeile, die in die andere Richtung gehen. Das ist hier für den Sprungbefehl und das ist hier praktisch das Ergebnis zurückschreiben ins Registerpfeil. Okay, hier haben wir nochmal die unterschiedliche Möglichkeit für Konstanten, was da passieren kann. eine sprungziel berechnung und dieses sprungziel berechnet sich mit hilfe dieser konstanten das ist der blaue pfeil hier und sie gehen dann gesehen dann geht es dann weiter auf dem program counter und das andere ist wenn sie im speicher mit einer immediate in der adresse zugreifen dann müssen sie natürlich eine adressrechnung machen und für die adressrechnung wird eben diese konstante auch auf 32 bit aufgefüllt und wird dann für die adressrechnung der alu zugeführt und der andere teil der adresse diesem Register 1 da oben, also aus dem anderen Source Register. Deshalb diese beiden Pfeile. Okay, das sieht jetzt schon sehr schön strukturiert aus. Wir können uns schon langsam vorstellen, dass wir eben einen Befehl nach dem anderen in diese Pipeline reinschieben und dass wenn eben der erste Befehl seine Instruction Fetch erledigt hat, wird er weiter geschoben nach Instruction Decode und der nächste ist dann im Instruction Fetch drin. gleichzeitig in Bearbeitung. So, ganz so einfach ist es natürlich nicht,

weil in jedem Taktzyklus verwende ich natürlich die komplette Hardware der Pipeline-Stufe und ich erziele ja in den einzelnen Pipeline-Stufen Ergebnisse und diese Ergebnisse muss ich natürlich immer von der Stufe N in die nächste Stufe weiter rufen und dafür brauche ich natürlich Pipeline-Register, die da eingebaut werden müssen. Das heißt, meine Maßnahme ist ganz einfach. Jeder Schritt, der durchgeführt wird und Ergebnisse produziert, diese Ergebnisse werden im Pipeline Register gepuffert und werden eben dann beim Takt praktisch übertragen in die nächste Stufe. Man kann sich das so vorstellen wie so ein Master-Slave-Flip-Flop, aber wir machen hier kein Master-Slave-Flip-Flop, sondern wir machen das mit Hilfe von Kombinatorik zwischen den einzelnen Speicherelementen. machen also zwischen die einzelnen Stufen unserer Pipeline diese Register rein. Das heißt, wir haben hier vier Stellen, wo wir Register einbauen müssen, also zwischen Instruction Fetch und ID, IDX und so weiter. Was ich natürlich nicht brauche von hinten nach vorne, also ich brauche den Kreisschluss nicht, von Writeback nach Instruction Fetch, da brauche ich keine Pipeline Register, also von der letzten Phase. Das heißt, von hier, hier brauche ich nicht nochmal Register, weil hier liegt ja das Ergebnis fest. Entschuldigung, ich brauche nur zwischen den Phasen hier Instruction Fetch ID, Instruction Decode und Execute, Execute und Memory und Memory und Writeback, hier diese blauen Stellen ist da, wo meine Pipeline Register angesiedelt sind und wir sehen es sind also 5 Befehle gleichzeitig in der Pipeline, Befehl N, der hier ist, der nächste ist der N plus 1, N plus 2, N plus 3, N plus 4. Diese fünf Befehle, die muss ich immer gleichzeitig beachten. Heute ist natürlich klar, das müssen wir jetzt dann im Folgenden noch machen, es kann immer passieren, dass da Datenabhängigkeiten auftreten und diese Datenabhängigkeiten führen zu Konflikten, die ich erkennen und lösen muss. So, wollen wir uns mal jetzt ein ganz konkretes Beispiel anschauen. also zum Beispiel den Befehl LOADWORD was in den einzelnen Phasen passiert erst mal wie schaut die Maschineninstruktion für ein LOADWORD aus wir haben hier oben in den obersten 6 Bits kodiert um welche Operation das es sich handelt wir haben hier kodiert praktisch in dem Register das adressiert ist in den Bits 25 bis 21 einen teil der adresse und wir haben hier hinten in den untersten 16 bits die konstante ja das ist praktisch die das gibt mir an welche das register ich nehmen muss um die adressrechnung auszuführen und das gibt mir an wohin ich das was ich aus dem speicher lese schreiben soll ins register file also das ist mit den bits 20 bis 16 adressiert in register transfer schreibweise so aus zugriff auf den speicher an der stelle 100 plus dollar zwei wird in dollar eins gespeichert also load word dollar eins 100 in klammern dollar zwei ist dann die schreibweise dafür und das dollar eins sind praktisch diese bits hier die in rt stehen und das dollar zwei das sind die bits die in rs stehen Gut, und jetzt wollen wir uns einfach mal den Durchlauf dieses Befehls durch die Pipeline anschauen und auch die Belegung der Register. Und deshalb haben wir hier unterschieden, also wenn links hellgrau ist, dann soll das Register beschrieben werden, wenn rechts hellgrau ist, dann soll das Register gelesen werden. Also, haben wir jetzt die erste Phase für unseren Load-Word-Befehl, ist die E-Fetch-Phase. wird eben hier aus dem Instruktionsspeicher gelesen. Das Ergebnis, was ich hier lese, speichere ich praktisch in den Registern Instruction Fetch, Instruction Decode. Das heißt,

im Wesentlichen wird hier die Instruktion, die ich lese, im entsprechend dafür vorgesehenen Pipeline Register zwischengespeichert, Befehl, dass der dann gleich geholt werden kann anschließend in der nächsten Phase. Gut, das heißt der Addierer berechnet die nächste Adresse, also BC plus 4 und im Normalfall, wenn dieser Multiplexer nicht geschaltet ist, steht dann der Befehlszähler gleich auf den nächsten Befehl für den dann geholt werden kann. Okay, jetzt haben wir dann die Instruction Decode Phase, das heißt ich den Pipeline-Registern hole ich mir die Instruktion, deshalb ist der rechte Teil grau gezeichnet und diese Instruktion wird praktisch ausgewertet. Ich hole mir also aus der Instruktion die Adressen für meinen Register-File, stelle also meine Operanten hier zur Verfügung und mit der Sign-Extension, wenn da eine Konstante drin steht in diesem Instruktionswort, wird diese Phase hier drei Ergebnisse, nämlich je nachdem, welchen Befehl ich habe, wenn eine Konstante auftaucht, dann brauche ich Design Extension, wenn ich einen normalen R-Teil Befehl habe, diese zwei Operanten und dementsprechend müssen die Ergebnisse dieser Phase in diese Pipeline Register hier reingeschrieben werden. Außerdem gebe ich immer noch weiter weil ich die später auch nochmal brauche, ins Pipeline Register zwischen ID und X. Das sind also die Ergebnisse, die hier zwischengespeichert werden. Das ist hier nochmal textuell erklärt da oben. Also Instruction Fetch ID stellt eben die Immediate aus der Instruktion zur Verfügung und beide Registerinhalte werden geliefert für die IDX-Phase. weil der Eingang nicht benötigt, weil das kommt ja dann eben aus der Sign-Extension. Okay, nächste Phase ist die Execute-Phase. Execute-Phase ist klar, greift wieder lesend auf IDX zu und schreibt in XMEM rein in diese Register. So, was Sie lesen, haben wir auf der Folie vorher besprochen, was eben an Ergebnissen zur Verfügung gestellt wird. Und was produziert jetzt die X-Phase an Ergebnissen? Einmal natürlich das ALU-Result. ergebnis der alu dann wenn die bedingung ausgewertet wird wird eben eine zero bit gesetzt oder nicht gesetzt dann wenn ich hier eine adress berechnung habe für eine sprungziel adresse dann wird natürlich diese sprungziel adresse hier berechnet muss dann eben entsprechend in dem register hier gespeichert werden und wenn ich hier diesen weg begehe dann muss natürlich dieses read data 2 weil ich das eben FNVL brauche, um auf den Speicher zuzugreifen oder einfach um direkt weitergeleitet zu werden. Gut, da haben wir nochmal Erklärung der Multiplexer. Der Multiplexer für den ALO Input 2 kann natürlich wählen zwischen Read Data 2 und dieser Sign Extension, logik dafür ok da können wir die nächste phase gehen nächste phase ist der speicher zugriff wir betrachten ja gerade beispielhaft den befehl load word und klar da lese ich auf dem speicher das heißt in dieser phase wird das was auf dem speicher gelesen wird eben zur verfügung gestellt wird wird entsprechend ins register memory write back ins pipeline register memory write back Und hier unten der Weg würde natürlich für normale R-Format Befehle gegangen werden. Also wenn jetzt kein Load vorliegen würde, sondern ein normaler arithmetischer Befehl, dann wird einfach dieses ALU-Ergebnis hier weitergeleitet in diese Memory Writeback Register. Okay, und in der letzten Phase, da haben wir ein Writeback. unseres Load-Wert-Befehls natürlich. Ich habe hier Hausnussspeicher gelesen, habe das ins Pipeline-Register MemWriteback gespeichert und was ich hier gelesen habe, wird über diesen

Multiplexer zurückgeschrieben ins Register-File, weil das Ergebnis vom Load wird natürlich im Register-File abgelegt. Okay. Gut. Was jetzt natürlich ein bisschen das Problem ist, haben wir ja vorhin schon mal darüber gesprochen, dass die Adresse im Registerfile an einer anderen Stelle beim Load liegen als bei einem R-Type Befehl. Wir müssen hier ein bisschen aufpassen, dass die Schreibregisteradresse richtig gestellt wird und müssen deshalb so eine kleine Logik einbauen, die uns unterscheidet, ob es jetzt ein Load Befehl ist oder ob das Ganze ein R-Type Befehl ist. gut wir haben noch einen kleinen fehler in unserem pipeline datenpfad den wir auch noch korrigieren müssen das machen wir dann zum beispiel hier da sieht man es dann besser als dass man sexuell erklärt also hier ist noch mal die tabelle für diese drei formate der befehle und die entsprechenden belegungen hier sind die bits in dem befehlswort und da müssen wir eben aufpassen es in rd abgelegt und beim speicherzugriffsbefehl in rt das heißt hier haben wir die 5 bits beim befehl wo auf den speicher zugreift und hier haben wir diese 5 bits und deshalb bauen wir diese kleinen multiplexer diese logik ein da wird eben unterschieden handelt es sich jetzt um eine ganz normale alu operation dann muss ich als schreibadresse für das registerpfeil rd nehmen also ein Speicherbefehl, dann muss ich als Schreibadresse für das Registerpfeil eben diese Adresse hier durchschalten. Und das wird natürlich wieder in den entsprechenden Registern zwischengespeichert und dann geht eben der Weg zurück. Und damit stelle ich sicher, dass die Ergebnisse, die hier zurückgemeldet werden, an die richtige Stelle gespeichert werden. erklärt ist. Und es ist natürlich klar, dass Pfeile nach hinten immer ein kleines Problem darstellen, weil wir haben ja so einen Datenfluss von links nach rechts und immer dann, wenn ich zurückgehe, muss ich natürlich aufpassen, dass ich nicht kollidiere mit den anderen Befehlen, die sich hier gleichzeitig in Bearbeitung befinden. Okay, was natürlich wichtig ist, ich kann eine Hardware zu einem Zeitpunkt immer nur belegen und dementsprechend so eine pipeline wenn man sich da überlegt wie viele unterschiedliche möglichkeiten der befehlsbelegung dass es gibt muss ich mir natürlich erstmal veranschaulichen ob ich denn meine ressourcen so aufgebaut habe dass die nicht doppelt belegt sind und muss einfach mal die unterschiedlichen befehlstypen durchspielen wie die diese ressourcen belegen um dann eventuell wenn ich ressourcen doppelt auslegen muss das zu erkennen und das kann man drei unterschiedlichen möglichkeiten machen man kann so einen tag snapshots machen wie wir es für log word befehl gezeigt haben auf den folien vorher kann man sich das einfach noch mal anschauen oder man arbeitet mit reservierungstabellen und da kann ich einmal die ressourcen über die zeit betrachten oder die befehle über die zeit betrachten da haben wir ein beispiel dafür für eine reservierungstabelle und zwar in dem beispiel haben wir die Das heißt, wir haben hier unseren Takt, wir haben hier unsere Ressourcen, also Instruktionsmemory, Registerfile, ALU, Datamemory und dann wieder Registerfile. Und hier haben wir eben die entsprechenden Phasen und die Belegung der Pipeline Register. den Load-Wordbefehl, der ist in grün gezeichnet und einen Subtraktionsbefehl. So und wir beginnen mit dem Load-Wordbefehl, das heißt im ersten Takt wird dieser Load-Wordbefehl aus dem Instruktionsregister geladen, also keine Kollision. Im nächsten Takt befindet sich praktisch der Subbefehl in dieser ersten Der greift dann auf den

Instruktionsspeicher zu, während das Load-Word ist schon in der ID-Phase und in der ID-Phase bewegt sich das Load-Word natürlich nur im Register-File. So, in der nächsten Phase ist die ALU belegt vom Load-Word-Befehl und der Subbefehl, also die Subtraktion, die beschäftigt sich mit den Registern. In der nächsten Phase beschäftigt sich der Subbefehl mit der ALU und der Load-Word-Befehl wieder mit den Registern. Entschuldigung, im Data Memory, Load-Word greift auf den Speicher zu, liest vom Speicher. der letzten Phase, da ist es jetzt dann so, dass das Load-Wordbefehl aufs Registerpfeil zugreift und im sechsten Schritt greift dann das Sub auf dem Register zu. Gut, also man kann sich das jetzt dann die Tabelle genau anschauen und schwierig werden würde es natürlich, wenn zwei Befehle nicht der Fall und dementsprechend kann man davon ausgehen, dass diese beiden Befehle halbwegs konfliktfrei erstmal durch diese Pipeline durchlaufen. Ob die jetzt dann Operanten haben, das heißt also wir haben hier keinen Check auf Datenabhängigkeiten, wir haben hier nur einen Check auf Ressourcen. Datenabhängigkeiten können natürlich trotzdem was sehen, die kommen dann später. Wir wollen hier nur überprüfen, ob ausreichend Ressourcen zur Verfügung stehen, Und das ist hier der Fall, weil eben keine Konflikte ersichtlich sind. Man kann das auch anders machen, indem wir einfach die Zeit laufen lassen entlang der Befehle. Das heißt, wir haben hier zwei Befehle, Loadword und Sub, genau die gleichen wie vorhin. Und hier schreiben wir dann einfach die entsprechenden Belegungen rein der Ressourcen und überprüfen dann, ob es dann irgendwelche Konflikte gibt. auftaucht. Also wenn hier zum Beispiel auch Instruction Memory stehen würde in Takt 2, dann hätten wir hier einen Konflikt. Solange hier in den Spalten praktisch nicht die gleichen Ressourcen auftauchen, haben wir keinen Ressourcenkonflikt und ich kann meine Hardware ebenso bauen, wie ich mir das vorstelle. Okay, man kann das Ganze natürlich noch für beliebig verfeinern, je nachdem wie komplex die Pipeline ist und welche Befehle man alles zulässt, um das Hier haben wir auch wieder die zwei Befehle Loadword und Subtraktion. Wir haben die einzelnen Taktphasen und wir haben eben noch zusätzliche Ressourcen, also zum Beispiel die Pipeline Register, die wir hier betrachten wollen, wie die belegt sind, um eben die Simulation möglichst realitätsnah zu machen und dann nachzuschauen, ob denn wirklich keine Ressourcen zu irgendeinem Zeitpunkt damit belegt werden. nacheinander ab und hier eben die entsprechenden Belegungen angezeichnet. Okay, so jetzt müssen wir uns überlegen, wie wir denn genau die Pipeline steuern, dass die Befehle korrekt abgearbeitet werden, nacheinander bzw. gleichzeitig bis zu 5 Befehle in der Pipeline, ohne dass wir uns da gegenseitig was kaputt schreiben, also wir möchten natürlich die korrekten Ergebnisse berechnen. Wir brauchen ein paar Erweiterungen, damit diese Pipeline funktioniert. Wir brauchen in der Instruction Fetch Phase brauchen wir Steuersignale WritePC und Read Instruction Memory. Da jeder Befehl in der Instruction Fetch Phase eben den Befehlszelle erhöht und neu schreibt und jeder Befehl eine Instruktion holt, sind diese beiden Steuersignale ein neuer Befehl geholt. So, im ID-Register, da werden die Operanten zur Verfügung gestellt in jedem Taktzyklus, da brauche ich keine Steuersignale, ich werde einfach meine Ausgänge vom Register-File aus. Wenn ich ins Register schreibe, dann muss ich natürlich Register Write, die

Steuerleitung entsprechend aktivieren. So, in der X-Phase, da haben wir ja jetzt Stressrechnung, die Bedingungen ausrechnen, die normalen Operationen und dementsprechend habe ich mehrere Möglichkeiten für die Quellen und ich brauche also so ein Steuersignal ALU Source für die ALU Quelle 2, wo ich dann auswähle zwischen der Konstanten, die auf 32 Bit aufgefüllt ist oder dem Register 2 und die ALU selbst muss natürlich angesteuert wie wir das vorher schon gemacht haben im Single Cycle Datenpfad. Wir verwenden eben 2 Bits für ALU-OB und 5 Bits, die aus dem Funktionsteil kommen, also 6 Bits, Entschuldigung, aus dem Funktionsteil vom Maschinebefehl. Also Sie wissen, in der Instruktion haben wir ja 6 Bit, die als FUNC bezeichnet werden, vorgesehen. Da steht im Wesentlichen drin, welche Operation die ALU ausführen muss. Wir haben im Instruktionswort diesen Operations-Teil ganz am Anfang, die 6 Bits, und von diesen 6 Bits ganz am Anfang verwenden wir aber nur 2 Bits. Wir haben das festgelegt im Single-Cycle-Datenpfad, da können Sie das nochmal genau nachlesen, wie diese Bitbelegung ist. So, und wir müssen natürlich noch die Zielregisteradresse unterscheiden, ob ich jetzt ein I-Format oder ein R-Format Befehl habe, und dafür gibt es ein Steuersignal Registered Destination. ja schauen wir uns das dann im, Entschuldigung es sind noch ein paar Erweiterungen drin, schauen wir uns das dann später im Bild an wir müssen im MEM, klar sagen, dem Datenspeicher müssen wir mitteilen, soll jetzt gelesen oder geschrieben werden das heißt wir müssen die Signale Memory Read und Memory Write korrekt ansteuern und wenn ich jetzt so ein Branch Befehl habe, dann haben wir ja dieses Zero Bit, das gesetzt wird um zu zeigen ob die Bedingung wahr oder falsch ist Das muss ich natürlich auch entsprechend verbinden. Beim WriteBack muss ich eben unterscheiden, ob ich in der ALO was berechnet habe, was ich ins Registerfall zurückschreiben will, oder ob ich mit Load auf den Speicher zugreife und das, was ich lese, ins Register zurückschreiben will. Deshalb gibt es den Steuersignal MemToRec, wenn ich aus dem Speicher was lese oder eben aus ALO Result. Okay, ich muss mir also so eine kleine Tabelle machen und muss mir für die unterschiedlichen Befehlsformate die Belegung der entsprechenden Steuerleitungen überlegen. Wir haben also hier R-Format Befehl, Load Befehl, Store Befehl und einen Branch Equal, also einen Sprung Befehl. Für IFID brauchen wir da nichts eintragen. Register Destination und ALO Source ist natürlich wichtig. also da ist kodiert bei 1 0 das ist ein r format befehl ist bei 0 0 ist es ein load befehl oder ein store befehl das macht ja für die alu keinen unterschied weil die alu hier eine adresse berechnet und für die adresse ist es ja letztlich egal ob gelesen oder gespeichert wird deshalb sind die alu opiz hier gleich und wenn ich ein branch habe dann wird es eben mit 0 1 in diesen beiden bits kodiert die roten steuern hier die multiplexer register destination und alu source die hier angezeichnet sind aber denken sie sich die einfach an den pfeilspitzen die wir haben und dann haben wir eben hier noch memory to register kommt auch normalen multiplexer dazu der angesteuert werden muss und zwar eben wenn ich ein r format befehl habe nehme ich ja das ergebnis aus der alu also nehme ich hier als steuerung die 0 für diesen multiplexer und beim load befehl natürlich wieder für don't cares so und beim r format befehl ist klar das ist kein sprung ich mache auch kein memory write ich mache kein

memory read aber ich schreibe ins register dementsprechend 0001 beim load kein sprung ich schreibe nicht auf dem speicher aber ich lese vom speicher und das Store ist auch kein Sprungbefehl und beim Store schreibe ich natürlich auf den Speicher, also Memory Write ist auf 1, Memory Read entsprechend auf 0 und ich schreibe auch nicht ins Register, sondern ich schreibe eben in den Speicher, deshalb steht hier die 1. Und beim Branch Befehl, da greife ich überhaupt nicht auf den Speicher zu, also haben wir hier lauter Nullen, aber das Signal für Branch ist natürlich auf 1. Also diese Tabelle ergibt sich ganz logisch durch genaues Hinschauen auf meinen Datenfaktor. So, unsere Main Control Unit war ja im Single Cycle relativ einfach, weil sie ja für den ganzen Zyklus gegolten hat. Jetzt haben wir ja 5 Befehle gleichzeitig in Bearbeitung, das heißt, ich muss die Signale, die aus meiner Main Control Unit kommen, praktisch mitführen mit meinem Befehl, dass die nicht verloren gehen. output der main control unit das heißt wenn mein befehl geladen wird dann werden diese ganzen steuersignale erzeugt und diese steuersignale werden eben in den pipeline register mit gespeichert damit ich die bei den einzelnen phasen durchführen kann also die main control unit berechnet alle steuersignale während der id phase und die werden dann schrittweise verbraucht und weitergeleitet. So, gut, also wir haben sichergestellt, dass wir ausreichend Hardware haben. Das heißt, wir haben keine Ressourcenkonflikte, so wie unsere Pipeline aufgebaut ist. Was aber passieren kann, das sind natürlich Datenabhängigkeiten, Datenkonflikte. Hier mal ein extremes Beispiel Das sind auch die einzigen Konflikte, die wir jetzt hier behandeln wollen. Und zwar haben wir hier fünf Anweisungen. Also eine Subtraktion, eine Endoperation, eine Oderoperation, eine Addition und ein Storeword. So, wir haben hier das Register $2, das in jedem dieser Befehle auftaucht und dementsprechend haben wir natürlich Datenkonflikte ohne Ende. 2 als Ergebnis berechnet. Bei dieser Endoperation wird es als Eingabeoperant verwendet. Das heißt, eigentlich darf ich diese Endoperation erst dann ausführen, wenn das Ergebnis in Dollar 2 liegt. Und dementsprechend, ich lese hier, nachdem ich schreibe. Deshalb heißt dieser Konflikt Read after Write. Und klar, ich muss jetzt dafür sorgen, dass das Dollar wäre, wäre das überhaupt kein Problem, weil erst wird diese Anweisung hier ausgeführt, dann liegt das Ergebnis vor in Dollar 2, dann wird diese Anweisung ausgeführt, kann ich also Dollar 2 verwenden. Wenn ich 5 Befehle gleichzeitig in der Pipeline habe, dann muss ich genau betrachten, ja wann liegt denn dieses Dollar 2 vor, damit ich das im nächsten Befehl verwenden kann. Und hier der Befehl verwendet auch nochmal Dollar 2 und das Add auch nochmal, habe dann muss ich erstmal schauen ob es ist denn tatsächlich ein konflikt oder liegt das ergebnis schon vor und ich kann es verwenden und wenn ich ein konflikt erkenne dann muss ich den natürlich irgendwie behandeln gut also ich muss natürlich meine register belegungen die ich jetzt in den pipeline registern habe betrachten und schauen ob da was drin steht was Das heißt, ein Konflikt besteht eben dann, wenn mindestens eines der Register, was in der FID-Phase ausgelesen werden soll, das gleiche ist wie eines der Register, die in der IDX, XMEM oder MEM-Writeback-Phase beschrieben werden sollen. Gut, also Read after Write, ich will hier was lesen, was eben hier geschrieben wird und muss warten, bis das tatsächlich geschrieben ist. Es

ist natürlich klar, dass ich solche Konflikte jetzt konstruiert habe und nicht zwischen jedem Befehl Konflikte auftreten, sondern nur dann, wenn tatsächlich Ergebnisse zurückgeschrieben werden. Sie erinnern sich, wir haben ja unsere schöne Datenflussrichtung von links nach rechts, also eigentlich kann da nichts passieren, aber wenn ich eben so ein Write-Back mache ins Register-File, dann gehe ich zurück, also Register-Write ist aktiv und dann kann eben nur so ein Konflikt auftreten. ok das heißt wir bauen jetzt so eine kleine logik um unsere konflikterkennung erstmal durchzuführen wir gehen aus von vom neuen befehl der befindet sich jetzt in der fid phase und in diesem befehl lesen wir also ein register und wir müssen jetzt schauen ob dieses register in dem anderen befehl beschrieben wird also in der nächste befehl steht ja in der idx phase der Also nicht nächstes, sondern Vorgänger. Der Vorgängerbefehl steht in der ID X, der Vorvorgänger in XMEM und der Vorvorvorgänger in MEMWriteback. Das heißt, wir müssen diese drei Befehle überprüfen, ob hier ein Write Register mit der gleichen Adresse auftaucht. Ich schaue also hier nach der Registeradresse, hier nach der Registeradresse und hier nach der Registeradresse, zwei Readregister ausfällt, dann habe ich einen Konflikt erkannt. Das heißt, ich brauche also 1, 2, 3 Befehle, muss ich weiterschauen mit meinem Konfliktfenster. Gut, jetzt muss ich eben für diese drei Situationen hier, für die und für die, muss ich die entsprechenden logischen Formeln aufstellen, die ich brauche, um diesen Konflikt zu erkennen. Erste ist jetzt der Konflikt zwischen X und ID-Phase. Wir müssen aufpassen, wir haben wieder das Problem mit den R-Format und Load-Befehlen, mit den unterschiedlichen Zielregistern, muss ich natürlich die entsprechende Unterscheidung machen, dass ich wirklich das Richtige miteinander vergleiche. Also eine Möglichkeit, der Befehl N ist in der X-Phase, werde aus ID und X, aus den Pipeline-Registern hier übernommen. Und der Befehl N plus 1 befindet sich in der ID-Phase. Dann muss ich natürlich auf die IF-ID, also Instruction Fetch ID Register zugreifen. Gut, schauen wir uns vielleicht die Tabelle dafür an, dann ist es, glaube ich, einfacher zu erkennen. Also wir machen einen Vergleich. IDX-Phase das Register beschreiben, ist die erste Abfrage. Wollen wir das beschreiben? Ja oder nein? Wenn wir das Register in der IDX-Phase nicht beschreiben wollen, dann haben wir kein Problem, dann haben wir keinen Konflikt, ist also die Konflikterkennung schon fertig. Wenn wir das jetzt beschreiben wollen, müssen wir unterscheiden, also wir sind hier im Ja-Teil Handelt es sich um ein R-Format Befehl oder handelt es sich um einen Load-Word Befehl? Beim R-Format Befehl müssen wir eben die entsprechenden Register 1, 2 vergleichen mit der Write-Register Adresse. Und zwar haben wir hier das RD für den R-Format Befehl, da ist das RD unser Destination Register. Also eben das ist hier ein Loadbefehl, da ist es RT. Deshalb brauchen wir nochmal diese Unterscheidung zwischen R-Format und Loadbefehl, damit wir hier die richtigen Register miteinander vergleichen. Das ist damit gemeint auf der Folie vorher. Und wenn sich jetzt eben herausstellt, dass eines dieser beiden Leseregister übereinstimmt mit der Schreibregisteradresse, dann haben wir einen echten Konflikt, dann haben wir einen RAW-Konflikt und können den entsprechend behandeln. ist dann haben wir keinen konflikt und hier ist der gleiche vergleich eben aber nur auf das rt nicht auf das rd wie hier also

auf die andere adresse in meine maschine befehlt und auch hier gilt wenn die schreibadresse die gleiche ist wie eine dieser leseadressen dann haben wir einen konflikt und müssen diesen konflikt entsprechend behandeln ich gehe noch mal eine folie zurück also das ist praktisch hier textuell ausgedrückt was wir dann auf der nächsten ok also das ist die konflikt erkennung die ich eben für die ex-id phase habe als ja wenn man so will als ablauf plan und aus diesem ablauf plan produzieren wir jetzt einfach logische quasi die Formel mit Logik, mit logischen Signalen. Wir haben hier uns und oders, das sind also logische uns, logische oders, die wir dann eben entsprechend setzen müssen. Also ein Konflikt liegt dann vor, wenn ich aufs Register schreiben will und einer dieser Fälle von der Folie vorher aufliegen. Ich gehe also hier durch, da haben wir eine 1, also ich will schreiben aufs Register, wir hier, es ist ein R-Format und ein Load-Word-Befehl und dann haben wir hier die nächste 1, es wird verglichen, Lese- und Schreibadresse ist die gleiche, also ist hier die 1 und dann haben wir also den Konflikt erkannt. Das heißt, wenn das Ergebnis dieser logischen Formel eine 1 ist, dann liegt ein Konflikt vor und wenn das Ergebnis dieser logischen Formel eben kein Konflikt vor. Das ist also eine einfache Logik aus ein paar Und- und Oder-Gattern, wo wir die entsprechenden Signale miteinander verbinden. Okay, damit haben wir also den Konflikt zwischen X und ID erkannt und ganz ähnlich läuft natürlich die Konflikterkennung zwischen Memory-ID. Ich habe genau dann einen Konflikt zwischen Memory und ID, in den Pipeline Register zwischen X und Mem, wenn die übereinstimmt mit einer Leseadresse aus der Fetch Instruction ID Phase, Leseregister 1 oder Leseregister 2 Adresse. Das ist also hier die logische Gleichung für den Mem-ID Konflikt und für den Write-Back Konflikt ist es eben auch entsprechend. auch wieder vergleichen die schreibregisteradresse in der writeback phase ob die übereinstimmt mit der leseregisteradresse in der id phase so und in der allerletzten phase also zwischen mem da brauche ich natürlich dann keine konflikterkennung mehr weil ich eine zyklenteilung vornehme das heißt in dem Zyklus gleichzeitig schreiben und lesen, kann also nichts mehr passieren. Gut, also das ist hier nochmal die Folie zur Zyklen-Teilung, das heißt, erst schreibe ich, wenn ich ein Ergebnis produziere zum Beispiel, in meinem Zyklus entteile ich in zwei Bereiche auf, in dem ersten Zeitraum, da wird geschrieben und im zweiten Zeitraum wird gelesen, Fall einen Konflikt einsparen, den ich dann nicht weiter betrachten muss. Das heißt, hier liegt dann eben kein Konflikt mehr vor, weil ich Zyklen-Teilung implementiert habe in diesem Befehl. Ansonsten wäre, wenn es keine Zyklen-Teilung gäbe, dann würde der Konflikt natürlich noch weitergehen bis zu diesem Additionsbefehl. Wie kann ich mit diesem Zyklus umgehen? Und eine Möglichkeit ist es natürlich, indem ich sogenannte Bubbles in die Pipeline einfüge. Das heißt, ich schalte einfach nicht weiter zum nächsten Befehl. Also mein Befehlszähler bleibt unverändert. Der wird nicht auf den nächsten Befehl erhöht. Und das kann ich ganz einfach dadurch erreichen, indem ich mit dem Steuersignal PCWrite verhindere, dass der Programme-Counter erhöht wird. schicke ich eine Bubble rein. Die andere Möglichkeit ist, dass ich eben den anhalte praktisch im ID, in der ID Phase, dass ich also ein if ID, Instruction Fetch ID, dass ich da 0 schicke als Signal. Das heißt also der Befehl wird einfach nicht gefordert, der bleibt einfach da und anstelle dass ich den Befehl forwarte

und diese null das sind natürlich ja umgangssprachlich wenn man so will das sind es bubbles also irgendwelche blasen die ich da in die pipeline einleitet die eben nichts tun sie erinnern sich an grundlagen der rechnerarchitektur wir haben da unterschieden zwischen stalls stall das ist so eine bubble also das ist wenn die hardware praktisch dafür sorgt dass nichts passiert dass nichts gemacht wird und knobs das sind no operations das sind also null operationen ein wird damit eben in einem befehl nichts gemacht wird beides ist natürlich möglich wir wollen uns jetzt hier darauf konzentrieren dass die hardware die probleme löst und nicht der compiler und dementsprechend sprechen wir von wabels oder stores ok diese stores die müssen wir natürlich irgendwie erkennen und einfügen und da brauchen wir den zusätzlichen multiplexer und die main control muss sie dieses dort natürlich generieren ja also die main control bekommt eine von der Konflikterkennung, da liegt ein Konflikt vor und du musst jetzt so und so viele Bubbles erzeugen. Das muss natürlich in die Hardware eingebaut werden. Das heißt, wir haben hier unsere Konflikt-Detection-Unit, die also die einzelnen Phasen überwacht, ob da ein Konflikt vorliegt und dann diesen Konflikt entsprechend meldet und dann als Ergebnis dann hier eben dafür sorgt, dass die Bubbles eingeleitet werden, Write-Signal und das If-ID-Write-Signal entsprechend auf 0 setzt und dann quasi die Nullen durch die Pipeline durchlaufen lässt und hier haben wir noch so einen Steuereingang für eine Bubble aus der Main Control Unit, der eben hier über die Conflict Detection Unit dieser Multiplexer angesteuert wird. Das heißt, was hier jetzt praktisch ist, das ist die ganze Hardware, die notwendig zu erkennen und bei Erkennung eines Konflikts dann dafür sorgt, dass so ein Stall ausgeführt wird. Okay, was natürlich unschön ist, wenn ich solche Leerzyklen erzeuge, ist, dass es kostet mich Leistung. Ich betreibe dann wieder eine Heizung, ich lasse die Pipeline laufen, aber die Pipeline macht eben in diesen Zyklen genau gar nichts. Und deshalb muss ich mir schon mal überlegen, wie kann ich denn so wenig wie möglich Bubbles einfügen, Und eine Möglichkeit ist, dass ich, wenn ich einen normalen R-Teilbefehl habe, dass ich meine ALU-Ergebnisse schneller verarbeite. Das heißt, dass ich da nicht erst warte, bevor ich die ins Registerfall zurückschreibe, sondern dass ich möglicherweise das Ergebnis, das ich berechne, gleich im nächsten Schritt, im nächsten Zyklus zur Verfügung stelle, der ALU und diese Abkürzung die wird in der Rechner Architektur forwarding oder bypassing genannt das heißt also ich berechne in der ALU ein Ergebnis und dieses Ergebnis wird vom ALU Ausgang direkt wieder auf den ALU Eingang zurückgeleitet. Diese Zurückleitung auf den ALU Eingang will ich natürlich nur in Konfliktfällen machen das heißt ich muss erkennen wann brauche ich das forwarding muss also so kleine Hardware bauen wieder, die das vorwärtigen erkennt die Situation und dann müssen die Multiplexer die jetzt unterscheiden, soll jetzt der Eingang von der ALO aus dem Registerpfeil bespaßt werden oder soll der Eingang von der ALO aus dem ALO Ausgang bespaßt werden, diesen Multiplexer muss ich natürlich korrekt mit den Steuern. Weist sich natürlich im Wesentlichen aus meiner Konflikterkennung. Die Konflikterkennung liefert mir die Information, wann so ein Forwarding gebraucht wird. Werde ich hier an dieser Stelle einfach aus. Und wenn ich die dann ausgewertet habe, dann kann ich entsprechend die Multiplexer ansteuern. Was wir also jetzt eingebaut haben,

ist der grüne Teil hier. eingänge ja wenn der multiplexer eben entsprechend geschaltet ist ich kann also auswählen ob ich das ergebnis am alu eingang 1 haben will oder am alu eingang 2 haben will und wenn ich jetzt einen load befehl habe dann greife ich ja auf den speicher zu und dann kann ich eben auch das load aus dem speicher kann ich direkt hier zurückschalten auf die alu also was ich praktisch mache im dass die Ergebnisse ins Register-File zurückgeschrieben werden und beim Forwarding werden eben die Ergebnisse direkt von der ALU-Ausgang oder vom Speicherausgang direkt wieder auf die Eingänge der ALU geschaltet. Und damit spare ich mir natürlich einige Bubbles. steuern heißen ALU SELECT A und ALU SELECT B und da haben wir eben also wenn ich einen normalen register eingang habe dann sind diese beiden steuersignale auf null wenn aus der vorher gehenden befehl praktisches ergebnis kommt also aus der ex mem phase dann ist es 0 1 das ist bei rtype befehlen und beim load befehlen vorherige befehlen mem writeback phase dann habe ich hier so hier wird eben noch mal am beispiel von alu select a die einzelnen phasen noch mal durchgespielt was da auftaucht also wir haben hier die ergebnisse unserer vorwärtigen logik die eben entsprechend true oder false liefern die signale und je nachdem steuern dann diese und der ALU SELECT A bestimmt praktisch den Eingang A. Kommt er jetzt aus dem normalen Register, wenn diese Belegung hier vorliegt, kommt der ALU Eingang A aus dem vorherigen Befehl oder aus dem vorvorherigen Befehl oder aus dem vorherigen Befehl. Diese Belegungen hier zeigen das eindeutig an und damit steuere ich eben die richtigen Operanten auf die ALU Eingänge. so wenn wir so eine vorwärte unit aufbauen ist klar dass die konflikt detection unit auf einmal viel viel einfacher wird weil ich ja deutlich weniger konflikte habe ja anstelle hier ein konflikt zu haben und bubbles einzuspeisen habe ich ja den konflikt gelöst und damit wird der größte teil dieser konflikt detection unit nicht mehr benötigt es bleibt noch ein bisschen übrig werden also wenn ich einen load ausführe und ich brauche im nächsten befehl das ergebnis des load dann schaffe ich das eben nicht zu lösen das problem ja das schauen wir uns zwei folien vorher in dem bild noch mal an das problem hier wenn ich ein ganz normale addition habe dann habe ich ja im nächsten zyklus das ergebnis vorliegen wenn ich aber ein load habe dann muss ich erst auf den speicher zugreifen und habe dann das ergebnis vorliegen das heißt beim load im falle eines ein Bubble übrig und den muss ich natürlich die Forwarding Unit einfügen. Das heißt ich muss die Situation erkennen, dass ich ein Load habe und nach dem Load der Operand in der nächsten Maschineninstruktion direkt gebraucht wird. Die eine Möglichkeit ist, dass die Forwarding Unit eben dann Leerzyklus einfügt, haben wir jetzt hier in der Hardware im Beispiel nicht eingebaut. Die andere Möglichkeit ist, ich fange das vorher ab, indem ich eben dafür sorge dass so ein Load nicht zu spät ausgeführt wird. Das heißt, der Compiler kann zum Beispiel umordnen und sinnvolle Befehle in diesen Bubble-Slot einfügen, indem ich die Reihenfolge verändere, ohne natürlich die Semantik meines Programms zu verändern. Okay, das war es, was ich Ihnen zum Kapitel Datenpfad erklären wollte. betrachtet. Einen Single Cycle Datenpfad mit dem Ergebnis, dass alle Instruktionen innerhalb eines natürlich relativ langen Taktes ausgeführt werden. Dann der Multi Cycle Datenpfad, dass ich meinen Datenpfad

in fünf Phasen aufteile und nicht jeder Befehl alle fünf Phasen durchlaufen muss, wie zum Beispiel so ein Jump Befehl braucht nur drei Phasen, der R-Type Befehl vier Phasen und nur der Load Befehl Und der letzte Ansatz war praktisch, dass ich meinen Single-Cycle-Datenpfad so transformiere, dass daraus ein Pipeline-Datenpfad wird, damit ich eben nicht einen Befehl gleichzeitig betrachte, sondern wenn ich fünf Pipeline-Stufen habe, fünf Befehle gleichzeitig betrachte. die Prinzipien und Mechanismen verstehen. Und ich glaube, Sie haben auch so einen Eindruck gewonnen, wenn diese Pipelines eben sehr, sehr lang werden, dass das auch beliebig komplex werden kann, insbesondere die Konfliktbehandlung. Also stellen Sie sich mal so eine 14-stufige Pipeline vor, wie groß da Ihr Konfliktfenster sein muss und wie aufwendig dann auch die Logik für Konflikterkennung und Behandlung sein muss. Okay, das war es für heute. Vielen Dank für Ihre Aufmerksamkeit .