Guten Morgen meine Damen und Herren, ja letzte Woche vor Weihnachten. Wir stecken in der Vorlesung Rechnerstrukturen, immer noch im Kapitel Komponenten, Superskalare Prozessoren. Datenpfad, also die Phasen des Datenpfads in Instruction Fetch, Instruction Decode und so weiter. Diese Phasen wollen wir uns anschauen und wollen uns für die einzelnen Phasen überlegen, was wir tun müssen, damit so ein Superskalader Prozessor einigermaßen effizient läuft. Wir stecken immer noch in der Instruction Fetch Phase, das heißt wir holen Befehle und das Problem beim Superskaladen Prozessor ist, dass ich natürlich nicht einen Befehl hole, sondern einen ganzen Block von Befehlen. stellt, dass Branches an der Stelle ein Problem darstellen, weil ich nicht weiß, welcher Befehl folgt auf den Branch. Wird der Sprung genommen oder wird der Sprung nicht genommen? Und dementsprechend müssen wir relativ viel Aufwand hineinstecken, eine halbwegs gut funktionierende Sprungvorhersage auch durchaus mit spekulativer Ausführung zu schaffen. Das nächste Mal haben wir uns erst einfache Prädiktoren angeschaut, die mir helfen, die Und haben dann geändert mit sogenannten zweistufigen adaptiven Prädiktoren. Und hier soll also die Vorhersage in zwei Stufen passieren. An dem Beispiel, wo wir das letzte Mal aufgehört haben, sieht man nochmal die prinzipielle Vorgehensweise. Ich habe zwei Komponenten. Ich habe einmal das Branch History Register. Das ist also hier. Dieses Branch History Register, da speichere ich Sprungmuster. nicht genommen. Dann verwende ich diese Information, um in eine Tabelle einzusteigen, als Adresse quasi für eine Tabelle. In dieser Tabelle habe ich in der Vergangenheit praktisch meine Prädiktoren, also meine Vorhersage-Bits aktualisiert. Wir haben ja gelernt, 2-Bit-Prädiktoren ist das Beste, was man eigentlich machen kann. Also mehr Bits bringen nichts, wenige Bits so einen Zweier-Eintrag in dieser Tabelle und es ist relativ einfach, wenn ich also jetzt vier Bits im Branch History Register zur Verfügung habe, dann kann ich so eine Tabelle mit 16 Einträgen aufbauen und ich kann nur einfach eintragen, okay, bei diesem Sprungmuster, da hat sich der Sprung, der nächste Sprung das letzte Mal so verhalten. Also habe ich hier meine Vorhersage-Bits und diese Sprungmuster verwende ich zur Adressierung, Natürlich wird dieser Predictor dynamisch aktualisiert, also ständig aktualisiert. Das heißt, der kann ja auch ein paar Mal daneben liegen, dann würden natürlich die entsprechenden Vorhersage-Bits korrigiert werden. So, das große Problem, das ich jetzt habe, ich habe einen begrenzten Platz in meiner Hardware und ich muss jetzt aus dem großen Lösungsraum mich für die günstigste entscheiden. Was kann ich alles variieren an dieser Idee? systemweit zur Verfügung stelle, das heißt jeder Sprung trägt sich hier ein. Oder das andere Extrem wäre, jede Sprungadresse bekommt ihr eigenes Brandschichteregister, dass ich also eine ganze Menge, dass ich für jeden Sprung die Sprungmuster speichere und dann verwende. Also das ist die eine Möglichkeit, dass ich entweder mit einem Brandschichteregister oder mit vielen Brandschichteregister arbeite, einfach um diese ganzen Sprungmuster hier abzuspeichern. Und die andere Möglichkeit ist die Pattern History Table. Also das ist hier meine Tabelle mit den Prädiktoren. Baue ich da jetzt eine systemweite Tabelle, wo alle Prädiktoren für alle Sprünge quasi korreliert drinstehen oder spendiere ich für jeden Sprung eine eigene Pattern History Tabelle, wo dann praktisch sprungspezifisch, sprungadressspezifisch

muss man sagen, die Bits drinstehen. Grundsätzliche Möglichkeiten spannen eben diesen großen Lösungsraum auf, das heißt, das haben wir hier auf der Folie vorher. Also das ist das, was auf der nächsten Folie da eben steht. Ich habe eine globale Branch History Register, wo sich alle eintragen und ich habe eine globale Pattern History Table, wo auch alle Sprungadressen sich eine Table teilen. überlagern und ich damit Korrelationen weder erkenne noch irgendwie auflöse, noch irgendwie für meine Zwecke nutze. Also einfach, aber blöd. Und das kann man jetzt natürlich beliebig ausbauen. S steht bei dieser Tabelle immer für Z, G steht für global und P steht für per, also per Adresse oder per Sprungadresse. Das heißt, der andere Fall, den ich praktisch sowohl für die pattern history table als auch natürlich für die für das branch history entscheide mache ich das jetzt pro adresse oder mache ich das ganze pro set oder mache ich das ganze global gut Fälle, die da aufgespannt werden, die wollen wir uns praktisch jetzt im Wesentlichen mal anschauen. Also ich sehe, ja okay, passt. Gut. Der nächste Fall, den wir betrachten, ist jetzt ein Schema, das verwendet immer noch eine globale Branch History Register. Das heißt, hier haben wir das große G stehen für ein globales Branch History Register. Table. Das heißt, für jede Sprungadresse gibt es hier eine eigene Tabelle. Hier stehen wieder meine Prädiktoren drin und ich verwende das Sprungmuster der letzten Sprünge, wobei eben letzte Sprünge, alle Sprünge gemeint sind, nicht ein spezifischer. Und dieses Muster verwende ich als Einstiegsadresse für diese Tabelle. Also dieses Muster ergibt mir praktisch die Zeile, in welcher ich in dieser Tabelle nachschauen muss. Und die unteren Bits der zu glauben, dass ich da 2 hoch 32 Tabellen ablege, was ja prinzipiell möglich wäre, wenn ich nur auf die Sprungadresse gehe, sondern man beschränkt sich halt immer auf eine bestimmte Anzahl von Bits aus der Sprungadresse. Also zum Beispiel, keine Ahnung, wenn man 4 Bits nimmt von der Sprungadresse, dann hat man 16 unterschiedliche Pattern History Tables, also 16 Tabellen oder wenn man 5 Bits nimmt, dann hat man 32 Tabellen. Also man nimmt einfach man dann diese Pattern History Tables. Und in diesem Beispiel GAP, G soll heißen Globales Branch History Register und ein Per P für Per Address Pattern History Table. Damit kriege ich natürlich schon einige Korrelationen raus, weil hier sieht man ja bestimmte Sprungmuster, die geht über alle Sprünge, aber dann eben die Prädiktoren sind dann spezifisch für Wenn ich jetzt die unteren 4 Bits nehme und es gibt mehrere Sprungbefehle, die sich in den unteren 4 Bits gleichen, dann kann es natürlich passieren, dass ich die falsche Pattern History Table erwische, bzw. eine Pattern History Table erwische, die eigentlich für einen anderen Sprungbefehl aufgebaut wurde, weil ich unterscheide halt nur die unteren 4 Bits der Sprungadresse. Ich einfach feststelle, wenn ich so einen GAP-Ansatz mache, dass der Aufwand dafür zu groß ist, also weil per Address ist immer großer Aufwand, pro Adresse so eine Tabelle aufzubauen. Und deshalb baue ich Sätze praktisch. Also ich modularisiere das Ganze ein bisschen und fasse halt viele Adressen zu einem Satz zusammen. Und wenn dann der Sprungbefehl aus diesem Adresssatz kommt, dann habe ich dafür eine Pattern History Table. register und jetzt haben wir halt nicht mehr pro sprung adresse so ein pattern history table sondern wir haben das ganze in sätzen zusammengefasst und haben pro satz so eine pattern history table und wenn

mein sprungbefehl aus diesem spezifischen satz kommt dann selektiere ich damit die pattern history table und mein branch history register gibt mir wieder die richtige zeile und hier habe ich die redaktion jetzt kann man natürlich und es ist in diesen gelben kästen hier immer jetzt kann ausrechnen, wie viele Einträge ich generiere, je nachdem wie viele Bits ich spendiere. Wenn ich zum Beispiel 10 Bits spendiere für diese Sets, dann habe ich 2 hoch 10 per den History Tables und jeweils 2 hoch 4 Einträge mit jeweils 2 Bit. Ich kann also ausrechnen, dass dieser Ansatz 32 KB plus 4 Bit verbrauchen würde. vorher, dann haben wir natürlich 2 hoch 24 Pattern History Tables und das ist einfach zu groß. Also das kriegt man nicht auf dem Chip so einfach unter, 2 hoch 24 Tabellen da aufzuzeichnen. Während so 32 Kilobit, das ist natürlich ein realistischer Wert, so etwas kriegt man unter Umständen hin. Gut, wenn man jetzt diesen zweistufigen, adaptiven Predictor vergleicht mit dem Kapitel vorher, also KN-Prädiktoren besprochen haben, dann kann man sagen, okay, so ein 4-2-Prädiktor ist im Wesentlichen das Gleiche wie ein GAS, also Global Branch History Register Per Set Pattern History Table mit 4, 2 hoch 10 und man verwendet eben die wundersten 10 Bits der Sprungadresse für die Anzahl der Pattern History Tables. So, jetzt wollen wir uns mal die andere Möglichkeit anschauen, dass wir das Branch History Register nicht nur eines nehmen, Register in eine Tabelle reinschreibt, also pro Sprungadresse praktisch, mit Speicher die Muster, die er produziert hat und dementsprechend pro Sprungadresse so ein Planchistery-Register hält. Das, was ich da natürlich schaffe, ist, dass ich, da ich mein privates Planchistery-Register Und was natürlich dazu passen müsste, dass ich auch entsprechend viele Pattern History Tables habe, wenn ich jetzt aber nur eine Pattern History Table zum Beispiel habe, dann habe ich natürlich ein Problem mit Interferenzen, weil eben diese Pattern History Table hat im Beispiel hier jetzt, wenn ich hier 4 Bits habe, nur 16 Einträge und es kann ja durchaus sein, dass zwei unterschiedliche Sprünge das gleiche Sprungmuster erzeugt haben bei ihren letzten Sprüngen. History Table. So, also das heißt, diese Variante wieder, dass man nur eine Pattern History Table nimmt, die ist wieder zu einfach. Wenn ich dann wieder für jeden Sprungbefehl eine Pattern History Table mache, dann ist das wieder nicht realisierbar, weil es zu teuer ist. Also ist die logische Konsequenz, dass ich das dann wahrscheinlich mit Sets wieder machen werde. favorisieren. Ich fasse also wieder meine Sprungbefehle in Sets zusammen und habe dann pro Set eine Pattern History Table. Der Ansatz ist natürlich der gleiche, ob ich jetzt dann pro Sprungadresse so eine Table habe, also sieht alles gleich aus, es bezieht sich halt, entweder ist so eine Table für ein Set zuständig oder so eine Table für einen einfachen Sprungbefehl zuständig. Der Unterschied ist im Wesentlichen im Aufwand, aber eben Methode. Die Methode ist die gleiche. Die Methode ist wieder die, ich habe eine Branch-Adresse und ich habe pro Branch-Adresse, habe ich so ein Branch-History-Register, wo ich also mein Sprungmuster hinterlegt habe und dann mit diesem Sprungmuster selektiere ich eben die richtige Zeile in meiner Pattern History Table und unterschiedliche Sprünge sollten normalerweise eben dann auch in unterschiedlichen Pattern History Tables landen. Gut, wenn ich Wenn ich also jetzt mit P, also per Address arbeite für die Pattern History Tables, dann habe ich überhaupt keine Interferenzen mehr. Und wenn

ich jetzt mit per Set arbeite, also PAS, also PAS nochmal liest sich, das große P steht immer für das Branch History Register, Interferenzen, wenn nämlich unterschiedliche Sprungbefehle auf die gleiche Pattern History Table zugreifen, weil sie sich eben nur in diesen unteren Bits nicht unterscheiden, aber weiter oben durchaus eben sich um unterschiedliche Adressen handelt. Das heißt, ich habe eine Per-Set-Branch-History-Table, also hier sind die ganzen Branch-Register per Set abgelegt und dann habe ich entweder eine Pattern-History-Table oder ein Set von Pattern-History-Tables, wo ich dann eben darauf zugreife. Klumpbefehle im gleichen Set herauskommen, wenn das kann, eben sowohl in der Pattern History Table, also auf dieser Seite passiert, als auch im Branch History Register, wenn die sich überlagern, ja gut, dann habe ich halt eine Korrelation, wir wissen ja das Schlimmste, was passiert, ist, dass ich eine falsche Prognose mache, diese falsche Prognose kann ich immer noch handeln, was mich die falsche Prognose kostet, ist Performance. Und was wir als Rechnerarchitekten generell für ein Problem haben, das spiegelt sich hier ganz deutlich, ganz großen Lösungsraum muss ich die für mich beste heraussuchen. Das heißt, ich muss im Wesentlichen wieder Aufwand und Ertrag vergleichen und normalerweise ist es so, in der Rechnerarchitektur, ich bekomme eine bestimmte Anzahl von Bits oder Space auf dem Die vorgegeben und sage, ok, da muss jetzt deine Sprungvorhersage-Logik rein, da muss deine Prediction-Logik reinpassen und dann versuche ich halt mit Bits, die mir da zur Verfügung stehen, das Beste rauszuholen. So, dafür machen wir erstmal so eine theoretische Abschätzung der Kosten. Und das sind einfach die Bits, natürlich kommen da auch noch Verbindungsleitungen dazu, aber erstmal, was brauche ich an Speicher? So ein Prozessor besteht ja immer aus zwei Bestandteilen, einmal speichernde Elemente und dann irgendwelche Logik, die zwischen diesen speichernden Elementen ist. sein, was uns weh tut und dann haben wir hier die Länge unseres Branch History Registers, die ist hier mit K angegeben und dann haben wir hier die Anzahl an Pattern History Tables, also im globalen Schema eine, im Per Set, da kommt es natürlich darauf an, wie viel Sets, das ich definiere, das kürzen wir mit P ab und wenn ich natürlich pro Sprungadresse eine habe, dann kürzen wir das eben entsprechend mit B ab. alle diese neuen Möglichkeiten, die mir so zur Verfügung stehen, einfach eine Kostenfunktion aufstellen. Das heißt, ich kann meine Hardwarekosten in Bits berechnen, indem ich einfach aufsummiere, wie viele Bits für den entsprechenden Ansatz verwendet werden. Und es ist klar, dass der höchste Aufwand entsteht natürlich beim PHP-Scheme. Das heißt, ich habe eine Branch sprungadresse und ich habe eine pattern history table pro sprungadresse also php ist natürlich mit abstand das aufwendigste was da passieren kann gut ja bringt es denn jetzt wirklich was wenn man rein theoretisch also in der praxis funktioniert es eh nicht weil man diesen großen aufwand gar nicht treiben kann bringt es denn wenigstens was dafür muss man jetzt einfach mal durch diese neuen Möglichkeiten jagen auf dem Simulator, dauert natürlich entsprechend lange und dann einfach mal die Ergebnisse vergleichen, damit man sieht, was bringt dieser Aufwand und lohnt sich dieser Aufwand. Es ist klar, eine generelle Aussage ist natürlich, je mehr Bits ich spendiere, desto besser wird meine Vorhersagegenauigkeit. k und wir wissen ja dass das die unterschiedlichen muster in meiner peter history

table sind das heißt ich habe immer zwei hoch ist die länge meine peter history table und natürlich je mehr solche muster ich speichere ja also es macht einen unterschied ob ich vier oder fünf bits dafür verwende für das branch history register desto genauer werde ich natürlich so jetzt schauen wir Wenn ich jetzt mit Integer-Programmen arbeite, also SpecInt ist da so eine Benchmark-Suite, die man sich da einfach laden kann, und dann stelle ich fest, ganz generell mal, dass die globalen Schemata besser arbeiten. Also kommt wahrscheinlich daher, dass sehr viele verschachtelte If-Dents da auftauchen, weniger Schleifen, und dafür ist ein globales Schema einfach besser geeignet, diese Muster auszuwerten für die Vorhersage. Wenn ich es mir anschaue, also Speck Floating Point zum Beispiel, dann werde ich feststellen, okay, das sind oft sehr wissenschaftliche Berechnungen mit sehr großen Matrizen, mit sehr großen Laufschleifen. Und dann finde ich einfach raus, dass die Per-Address-Schemata besser arbeiten. Gut, da gibt es natürlich eine schöne Möglichkeit, viele wissenschaftliche Papers und Doktorarbeiten darauf zu verschwenden. detail anschauen kann was für tests die alle gefahren haben und was sie alles miteinander verglichen haben und die haben sich dann auch mal so in diesem paper so eine aufgabe gestellt und sagt okay wir haben jetzt ein hardware budget von 8 kilobit ja was ist denn die beste lösungsmöglichkeit für zwei stufigen adaptiven prädiktor um praktisch die höchste vorhersagegenauigkeit mit diesem aus, dass ein K gleich 6 und P gleich 16 das beste Ergebnis liefert für, und das ist ja immer lastspezifisch, für Speck 89. Das kann durchaus sein, dass es für andere Benchmark-Programme, für andere Lasten, haben wir ja vorhin gesehen, Unterschied zwischen Integer und Float-Lasten, dass es wieder ein bisschen anders ausschauen kann. Und es ist natürlich ein bisschen konstruiert, also wenn man sich dieses Schema anschaut, was kostet es, also P per Address Branch Set Pattern History Table, ja das K und P haben wir hier dann nochmal, also wir haben K gleich 6 und P gleich 16, dann kann man den Aufwand berechnen und dann kommt man eben zufällig auf die 8 Kilobit, natürlich nicht zufällig, ist ein bisschen konstruiert. So, wenn ich jetzt dann 128 Kilobit habe, ja, dann, und das ist vielleicht ein Ergebnis dieses Papers, mit K gleich 13 und P gleich 32, das Beste mit einer sehr schönen Genauigkeit von 97,2%. Soll also heißen, wenn ich jetzt mehr Bits spendiere, dann heißt es nicht unbedingt, dass das gleiche Schema, hier haben wir ein PAS-Schema und hier haben wir jetzt auf einmal ein GAS-Schema als Testsieger, dass das gleiche Schema bei unterschiedlichen Größen gleich gut funktioniert. spendiere, dass ich dann auch ein anderes Schema brauche. Und das muss ich eben mit Hilfe von Simulationen entsprechend herausfinden. Und das Ergebnis von tausenden von Simulationen, das ist das dann, was in die reale Architektur einfließt. Und da haben wir zum Beispiel so einen Alpha-Prozessor von DEC, der hat mit einem SAG-Schema gearbeitet, mit eben entsprechend K gleich 10 und P gleich 10 und hat das Ganze noch ein bisschen verfeinert, Das ist ein zweistufiger adaptiver Predictor und das war ein relativ erfolgreiches Sprungvorhersage über einen ganz großen Bereich von unterschiedlichen Lasten. So, jetzt kann man sich natürlich immer wieder überlegen, ja wie kann ich denn die Idee noch ein bisschen verfeinern. Informationstechniker oder Elektrotechniker, also wir Informatiker, wir liefern die guten Ideen und diese guten Ideen, die müssen ja dann irgendwann

so in Hardware umgesetzt werden und das sehe ich eher so als Aufgabe der Informationstechniker oder Elektrotechniker, die versuchen dann jedes einzelne Bit einzusparen und die Ansätze noch ein bisschen zu verändern. Den kann ich als G-Share-Prediktor realisieren und zwar was ich dann mache, dass ich die Bits des Branch History Registers, die ich gespeichert habe im Branch History Register und die Bits meiner Sprungadresse mit XOR verknüpfe, um das dann als Einstieg zu nutzen in eine Pattern History Table. dann aus dem Adressteil des Branch-PC und aus dem Inhalt des Branch-History-Registers baue ich dann durch Konkatenation, dass ich die ineinander schreibe, eben entsprechende Adressen auf. Also hier haben wir die XOR-Verknüpfung und hier haben wir die Konkatenation in diesem Beispiel. Also das ist dementsprechend der G-Select 4.4. Das heißt, ich nehme 4 Bits aus der Sprungadresse meines Sprungs 4 Bits aus dem Branch History Register und konkateniere die entsprechend. Oder hier haben wir diese G-Share-Lösung, dass ich eben die 8 Bits jeweils vom Branch History Register und meiner Sprungadresse mit XOR verknüpfe und das dann wieder als Einstieg verwende in meine Pattern History Table. Solche Sprungvorhersagen finden auch tatsächlich in real existierenden Prozessoren Verwendung. mit solchen Schemata. Gut, jetzt ist natürlich wieder die große Frage, was ist denn jetzt besser? Wer kann Interferenzen besser vorhersagen oder schlechter vorhersagen? Und dann kann ich praktisch nochmal mein G-Select und G-Share nochmal wieder ein bisschen verändern. Also das ist dann schon gehobene Stufe, also Stufe 3 praktisch. nehme die Erfahrungen, die ich bis jetzt gewonnen habe und versuche das Ganze nochmal zu verfeinern. Und da gibt es dann den entsprechenden Ansatz vom McFarling, der dann entsprechend beide Ansätze miteinander kombiniert technik also er macht hier ein ex-or vom branchister register und den adress präsentiert und verbindet das ganze so diese grundsätzliche idee dass ich in der ersten stufe quasi eine auswahl treffe und dann in der zweiten stufe auf die entsprechenden prädiktoren zugreifen kann man noch weiter verfallen da gibt so einen sogenannten Hybrid-Prediktor und der Hybrid soll heißen, er arbeitet nicht nur dynamisch, alles was wir bis jetzt gesehen haben, arbeitet dynamisch. Das heißt, das ist alles in Hardware, da kriege ich als Programmierer nichts davon mit, da kriegt auch der Compiler-Bauer nicht großartig davon mit. Das macht alles komplett die Hardware, also das ist eine hardwarebasierte Sprungvorhersage. Jetzt kann man sich natürlich überlegen, naja, will ich denn den Compiler ganz außen vor lassen oder will ich ein bisschen den Compiler auch mitspielen lassen? Und dann könnte man sagen, okay, wir machen das in zwei Stufen. Wir machen die Warm-up-Phase und in dieser Warm-up-Phase arbeitet der Compiler an statischen Vorhersagen. das Ergebnis der Bedingung Vorhersagen und damit auch Vorhersagen, ob der Sprung genommen wird oder nicht. Also einige Sachen kann man einfach statisch machen. Und warum soll man diese Information nicht verwenden? Und dann wechselt man natürlich dann auf der Hardware in die dynamische Vorhersage. Es ist ja auch immer so ein kleines Problem bei diesen ganzen dynamischen Vorhersagen. Ich muss ja meine ganze Datenstruktur erstmal füllen. füllen und das schaffe ich dadurch, dass ich die Sprünge beobachte und dann die entsprechenden Einträge mache und aktualisiere. Und wenn ich da eine gute Startposition habe, in dem ich vom Compiler schon ein paar Infos

bekomme und die dann entsprechend eintrage, dann hilft mir das natürlich. Die andere Möglichkeit ist, dass ich versuche, so einen Konfidenzintervall, also so einen Vertrauensintervall aufzubauen, dass ich also versuche, Wenn ich eine Laufschleife habe, dann vertraue ich einfach mal darauf, dass der Sprung relativ häufig genommen wird. Und wenn ich die Erfahrung habe, dass bestimmte Sprungsituationen dazu führen, dass der Sprung sehr oft wechselt zwischen genommen und nicht genommen, dann kann ich den klassifizieren als schwer vorhersagbar. Und es ist dann ein Low-Confidence-Branch, also ein Sprung, in dem ich wenig Vertrauen habe, eine gute Vorhersage zu treffen. high confidence branche das heißt ich weiß aufgrund irgendwelche informationen das weil es zum beispiel eine laufschleife ist und weil ich zum beispiel weiß die laufschleife wird 10.000 mal durchlaufen dann weiß ich okay mit großer wahrscheinlichkeit wird dieser sprung genommen werden und dann kann ich dann als high confidence branche also man versucht einfach jedes stückchen informationen dass man irgendwie nutzen kann um die sprungvorhersage zu verbessern versucht So, und dann kann man natürlich seine Ergebnisse immer wieder testen. Das heißt, wir haben hier Lasten, also unterschiedliche Anwendungen, unterschiedliche Programme, Compress, GCC, Go, also ein Spielprogramm, ein bisschen Integerverarbeitung, ein bisschen Floating-Point-Verarbeitung und dann schaue ich mir einfach, was passiert. So, und dann sehen wir, dass das schon größere Programme sind. programm die da auftauchen und hier haben wir die bedingten sprünge und da sehen wir im mittel sind 17 prozent aller befehle sind sprungbefehle also wissen wir ja schon und wir wissen dass ich es lohnt sprungvorsorge zu machen dann haben wir im mittel 54 prozent aller sprünge werden genommen das heißt 57,7 prozent 45,7 prozent werden nicht genommen und jetzt kann man Es gibt zwei Größen, die eben interessant sind. Das ist die Fehlerprognoserate und davon die beste und davon die schlechteste. Und da sehen wir, es gibt also so friedliche Programme wie dieses Vortex-Programm hier zum Beispiel. Da ist es also so, dass ich hier eine sehr geringe Fehlerprognoserate habe. Und wenn ich diese McFarlane-Kombination aus den beiden nehme, dann bin ich bei 1,7% Fehlerrate. Also das ist schon eine sehr, sehr gute Vorhersage und die hilft natürlich, den Prozessor effizient mit Befehlen zu versorgen. So, was natürlich nicht so gut ist, ist zum Beispiel das Go-Programm. Also das ist die Spielesimulation und da sehen wir, dass wir schon ganz schön daneben liegen mit unserer Vorhersage. Das heißt, wir haben eine Fehlprognoserate je nach Prädiktor zwischen 25 und 34%. Das ist natürlich tödlich und das zeigt, dass ein Superskalader-Prozessor für diese Last nicht so gut geeignet ist offensichtlich. die ich dann aber nicht in der letzten Phase committe, also als Ergebnis übernehme. Das heißt, diese Befehle belasten natürlich meine Hardware, die werden zugewiesen, aber die werden letztendlich weggeschmissen und das ist natürlich auch schon ernüchternd, dass ich 20 bis 100 Prozent der Befehle bearbeite, aber das Ergebnis nicht übernehme und die dann wegschmeiße. Also so eine superskalare Architektur ist eben nicht ohne und deshalb ist es schon sehr, sehr wichtig, Fetchphase so viel wie möglich korrekte Befehle zu machen. Also der Aufwand lohnt sich. Okay, das heißt, wir sind immer noch in dieser Instruction Fetchphase und überlegen uns, was können wir denn außer Prediction noch machen und dann kommt die Predication ins Spiel.

Also nochmal ein Unterschied zwischen Prediction und Predication. Prediction heißt übersetzt Vorhersage, Predication heißt übersetzt mit einem Prädikat versehen. Das die Informationen, die wir FNWL haben, nutzen, dem Befehl ein Prädikat mitzugeben. Und in diesem Prädikat steht im Wesentlichen drin, okay, die Wahrscheinlichkeit dafür, dass dieser Sprung genommen wird oder dass dieser Befehl an der Stelle gebraucht wird, ist hoch oder ist eben nicht hoch. Ich baue also bei jedem Befehl, spendiere ich ein paar Bits, um so ein Prädikat mit einzubauen. Und die Idee, die ich weiterhin habe, ist, dass ich nicht spekulativ einen Sprung ausführe, sondern dass ich spekulativ beide Richtungen weiterverfolge. arbeite ich an der Stelle weiter, wo ich den richtigen hatte und den Rest schmeiße ich wieder weg. Also einfach eine Ausführung von beiden Möglichkeiten. Und ich kann diese Ausführung ja nicht immer grundsätzlich ganz gleichzeitig machen, das heißt, ich muss dann eine gewisse Reihenfolge auch einhalten und da kann ich entweder in der Variante 1 diese Predication verwenden, das heißt, der Compiler generiert irgendwelche Prädikate, die Architektur, Paulin Hart, wer auch, arbeitet auch an diesen Prädikaten, Ich bringe da Informationen unter und das bestimmt mir dann die Ausführungsreihenfolge. Oder ich lasse den Compiler außen vor und bleibe wieder auf meiner Hardware-Ebene, also mache Mikroarchitekturverfahren und führe einfach mal stumpf immer beide Möglichkeiten aus, egal ob da jetzt ein Prädikat da ist oder nicht. Also da brauche ich dann nicht unbedingt ein Prädikat. dass ich jeden Befehl um Vorhersagebits erweitere und eben diese Vorhersagebits eine Auswirkung darauf haben oder eine Information enthalten, wie groß die Wahrscheinlichkeit für Taken oder Non-Taken Branches ist. Dass ich aber trotzdem, also das ist eine Möglichkeit für die Reihenfolge, dass ich aber trotzdem beide Seiten ausführe, ausführe und dass ich dann aber eben wenn die sprungentscheidung da ist dass ich einen zweig verwerfen muss ich ja machen weil wenn ich den falschen zweig weiter verfolge dann für das zu falschen ergebnissen und ich muss natürlich auch aufpassen dass ich keine seiten effekte einstellen das heißt ich muss da genau drauf passen was habe ich bis jetzt alles verändert und das muss ich alles zurück drehen und es kostet natürlich das ist teuer Und ich kann mir jetzt natürlich schon überlegen, pro Befehl Bits dazu zu bauen, das ist ja aufwendig. Und deshalb schaue ich mir einfach mal an, welche Befehle versehe ich denn mit Prädikaten. Und das kann ich entweder voll machen, dass ich also jeden Befehl mit einem Prädikat versehe, oder dass ich nur bestimmte Befehle, also zum Beispiel Load-Store-Operationen mit einem Prädikat versehe. Und da gibt es dann eben die unterschiedlichen Ansätze in der Realität, verfolgt da und die IA64 Architektur, also Itanium haben wir ja schon drüber gesprochen, die verfolgen so eine Fully Predicative Strategie, das heißt jeder Befehl wird mit einem Prädikat versehen und die Alpha, MIPS, PowerPC und Spark, die haben nur für die Load Befehle praktisch so ein Prädikat dabei. Ja, hier so ein Beispiel, wie man was mit einem Prädikat versehen kann, also das ursprüngliche Ich habe hier eine Abfrage, wenn x gleich 0 ist, dann führe diese zwei Instruktionen aus, auf jeden Fall führe aber diese Instruktion aus. So, das heißt, und danach kommen natürlich lauter Instruktionen, die unabhängig sind vom Branch B1. So, wenn ich jetzt mit Sprungvorhersage arbeite und liegt da falsch und habe schon diese Instruktionen a gleich b plus c und d gleich e

minus f ausgeführt, das natürlich löschen, also muss ein Rollback durchführen, dass da keine falschen Ergebnisse entstehen. Wenn ich das Ganze predikativ übersetze, also predikative translation, dann heißt es, irgendwann wird natürlich das Prädikat gesetzt, in dem Fall wird das Prädikat gesetzt, dann, wenn x den Wert 0 hat und dieses Prädikat, das steht ja jetzt im Befehlswort schon drin, plus c und die Instruktion d gleich e minus f ausgeführt und diese Instruktion g gleich h mal i, die wird natürlich immer ausgeführt. So, was ist jetzt eigentlich der Unterschied zwischen diesen beiden Darstellungen? Das sieht ja ziemlich ähnlich aus. Ich habe hier ein f und ich habe hier ein f. Der Unterschied ist der, dass ich hier dieses x gleich 0, diese Bedingung erst auswerten muss und diese Bedingung, und abhängig von dieser Bedingung ist diese Bedingung, wird dem Prädikat zugewiesen und im Prädikat steht jetzt drin, ob diese Instruktion ausgeführt wird oder nicht ausgeführt wird. Das heißt, in dem Sinn habe ich keinen Sprung mehr. Ich habe also den Sprung aufgelöst. Und das ist eben der Vorteil oder die Konsequenz von Prädication, ist der verzweigte Sprung fällt weg. Ich brauche keine Bedingungen mehr auswerten, sondern ich muss das Prädikat nur überprüfen. Da wird aber eben das Ergebnis nur übernommen, also das Commitment am Ende meines Datenpfades, wenn eben dieses Prädikat true ist. Das heißt, ich kann also diese Befehle auf jeden Fall erstmal ausführen und kann eben sicherstellen, dass das Ergebnis nicht übernommen wird. Der Befehl 4, klar, wird immer ausgeführt und das Ergebnis wird dann eben auch übernommen. Wenn jetzt mein Prädikat falsch war, dann habe ich natürlich Befehle 2 und 3 umsonst ausgeführt. richte ich auch keinen Schaden an. Das Einzige, was ich an Schaden anrichte, ist, dass ich halt Ressourcen verschwende. Ich habe also Register und Execution Units belegt, habe irgendwas berechnet, was ich dann wieder verwerfe. Gut, wenn meine Sprungvorhersage natürlich korrekt ist, dann, also wenn ich das Prädikat auch richtig setze, dann werden die Ressourcen tatsächlich nur benutzt, wenn sie sich auf dem korrekten Pfad befinden. Thema Producation, was sind jetzt die Vorteile? Der verzweigte Sprung fällt weg, ist als natürlich leichter zu handeln, wenn ich Befehlsbrücke laden muss. Ich brauche also in der Instruction-Fetch-Phase da jetzt nicht darauf zu achten, dass da ein verzweigter Sprung ist, sondern ich lade den ganzen Block von Befehlen für die aus. Ich habe natürlich in dem Sinn keine falschen Spekulationen mehr, weil ich keinen Rollback machen muss. Das heißt, ich fange das Ganze am Ende ab in der Commit-Phase. Ich committe das Ergebnis nicht und damit ich auch kein rollback machen so ein verzweigter sprung hindert ja immer ein basis block dran zu wachsen ja wenn sobald ich einen verzweigten sprung habe ist ende des basic blocks diese basic blocks also basis block auf deutsch basic block in englisch diese basic blocks die braucht der compiler die verwendet er auch für optimierungen also er macht praktisch erstmal mit BASIC-Blocks und innerhalb dieser BASIC-Blocks kann ich dann eben Optimierungen ausführen. Und je länger so ein Basisblock ist, desto mehr Möglichkeiten habe ich da. Und außerdem hat sich eben gezeigt, dass diese Predication gut ist für If-Then-Blöcke, also für Verzweigungen außerhalb von irgendwelchen Laufschleifen, die auch noch schwer vorherzusagen ist. Hier ist nochmal, hier unten im gelben Bereich steht nochmal die Definition von BASIC-Block. Jetzt natürlich, an welchem Punkt wird ein

Befehl gestoppt, wenn das Prädikat falsch ist. Ich habe bis jetzt immer gesagt, okay, es wird hier gestoppt, in der Commitment-Phase, ganz am Ende, Retire and Write Back, hier wird ja das Ergebnis committet. Und ich habe ja hier letztmalig die Möglichkeit, praktisch ein falsches Ergebnis zu verhindern, indem ich den Befehl oder das Ergebnis des Befehls einfach nicht committe. Das ist die Variante 2, also der Prädikative Befehl wird ausgeführt, Es gibt aber noch eine andere Möglichkeit, das ist die Variante 1. Ich behandle dieses Brett einfach wie einen zusätzlichen Operanten und das heißt, irgendwann hier wird dieses Brett schon ausgewertet. hier noch die Möglichkeit, ihn hier zu verwerfen, dann wird er eben an der Stelle gestoppt und nicht ausgeführt. Das sind die zwei Möglichkeiten, die ich habe und die sparen mir eben dann mithilfe dieser Predication das doch relativ aufwendige Rollback. Also egal, ob ich jetzt dann am Ende committe oder nicht committe, natürlich committe ich nur den korrekten Pfad. Aber wenn ich Spekulativbefehle ausführe, dann kann ich mir natürlich überlegen, in welcher Reihenfolge führe ich denn Spekulativbefehle aus. Schauen wir uns mal so einen Verzweigungsbaum an. Wir haben hier das Branch Level 1, habe ich zwei Verzweigungen im nächsten Branch Level, Also das verdoppelt sich von Level zu Level. Und natürlich kann ich alle Branches ausführen und das Ergebnis eben nur übernehmen, wenn das Prädikat wahr ist. Das ist aber, wenn man sich so ein Bäumchen anschaut mit exponentiellem Wachstum, exponentielles Wachstum, bis mal was es momentan ist durch Corona nochmal, dann wird es natürlich schon sehr aufwendig. Wenn also nur ein Pfad hier durch korrekt wäre und ich aber alle Pfade berechne, dann habe ich natürlich auch ziemlich viel Blödsinn gerechnet, den ich dann wegschmeißen muss. Ich kann natürlich sagen, okay, wenn ich theoretisch Ressourcen keine Rolle spielen, dann kann ich mit Eager Execution dieselbe Performance erreichen wie die perfekte Sprungvorhersage. Das heißt, da ich ja gleichzeitig immer alles berechne, Stichwort unbegrenzte Ressourcen, habe ich auf jeden Fall auch den richtigen Pfad in der kürzesten Zeit berechnet und bin also genauso gut wie eine perfekte Sprungvorhersage. von sehr viel ressourcen das ist natürlich nicht die praxis ja das ist wie physik ohne reibung in der praxis habe ich natürlich sehr limitierte ressourcen und der bedarf an ressourcen steigt exponentiell mit jedem jäger ausgeführten branche ist klar wir verdoppeln uns auf jeder stufe exponentielles wachstum und das haben wir einfach an ressourcen nicht vorhanden das heißt ich muss zu verdoppeln. Und da gibt es dann die Predictive Execution, das ist, ich führe auch spekulativ aus, aber ich versuche mit Wahrscheinlichkeiten zu arbeiten, entweder über Prediction Bits oder über andere Informationen kann ich Wahrscheinlichkeiten berechnen und je wahrscheinlicher eine Spruchrichtung Sprungrichtung und dann später die nicht so wahrscheinliche Sprungrichtung. Und wenn jetzt meine Vorhersage korrekt war, dann wird das Ergebnis übernommen. Sonst muss ich halt wieder so ein Rollback machen. Gut. Das Dümmste, was man machen kann, ist die Lazy Execution. Bei der Lazy Execution mache ich keine spekulative Ausführung von Branches. Und dann kosten mich aber Branches halt so richtig praktisch den ganzen befehlsfluss jeder branche und es spricht den befehlsfluss ich hatte mir jede menge von babels ein und bin dann entsprechend langsam gut wir wollen jetzt so eine multipfad ausführung ein bisschen genauer betrachten also wir führen auch

unnötige arbeiten aus und zwar die execution und die hardware lädt natürlich beide pfade taken und non taken obwohl nur einer Wir machen das in Hardware, also ein Mikroarchitekturverfahren. Wir lösen Konflikte natürlich so weit wie möglich auf, indem wir Registerkonflikte durch Umbenennung auflösen. Also wir gehen davon aus, dass wir genügend Register haben und dann einfach hardwaremäßig diese Registerzuordnung auch nochmal umbenennen können. Und wenn ich das jetzt wirklich alle Möglichkeiten immer ausführe, dann bräuchte ich natürlich eine 100% korrekte Sprungvorhersage. Haben wir gerade festgestellt, alles ausführen geht nicht, Ressourcenverbrauch, das heißt wir müssen irgendwie uns eine Reihenfolge erstellen, in der wir diese Sprünge ausführen wollen. Und natürlich ist meine Sprungvorhersage, je besser, desto höher die Genauigkeit meiner Vorhersage ist, das haben wir ja auch schon festgestellt. ist natürlich, dass ein Branch ist ja nicht ganz so schlimm, aber wenn die geschachtelt auftauchen. Das heißt, innerhalb meiner spekulativen Pfad kommt der nächste Branch und dann kommt wieder der nächste Branch. Und dann habe ich ja Wahrscheinlichkeiten hintereinander geschaltet praktisch. Also die Wahrscheinlichkeit für den nächsten Branch hängt natürlich von der Wahrscheinlichkeit des vorhergehenden Branches ab. Und deshalb habe ich drei Möglichkeiten jetzt zu spekulieren, welches nehme ich denn als nächsten. ich betrachte erstmal nur die Pfade, die eine höhere Wahrscheinlichkeit haben, oder ich ignoriere das einfach und betrachte alle Pfade, dann bin ich bei der Breitensuche, oder ich versuche die Reihenfolge zu optimieren mit einem sogenannten Disjoint-Eager-Verfahren, das aber nicht ganz einfach zu implementieren ist, ich muss nämlich für alle Branches Vorhersagen treffen, Ausführungswahrscheinlichkeiten über alle Pfade, die bis jetzt noch nicht ausgewählt wurden. Und dann wähle ich halt immer den Pfad mit der höchsten kumulativen Ausführungswahrscheinlichkeit. Das heißt hier, das sehen wir gleich auf der nächsten Folie, hier muss ich ein bisschen was tun, um dann die Entscheidung zu treffen, welchen Branch ich als nächstes lade, während bei diesen beiden Möglichkeiten hier ist es relativ einfach. Hier sehen wir nun diese drei Möglichkeiten. Wir sehen hier die Pfade, die möglichen Pfade versehen mit Wahrscheinlichkeiten, also 0,7% Wahrscheinlichkeit, das heißt 70% Wahrscheinlichkeit, dass es in diese Richtung gegangen wird, 30% Wahrscheinlichkeit in diese Richtung. Diese 70% hier teilen sich nochmal auf in 49% und 21%, dann die 49% teilen sich nochmal auf in 34% und 15% Wahrscheinlichkeit und so weiter. So ist das Ganze zu lesen. gehe ich einfach mal ganz konkret in diese Richtung nach unten. In dieser Reihenfolge führe ich also die Befehle aus. 1, 2, 3, 4, 5, 6. Das heißt, erstmal den Befehl mit der höchsten Wahrscheinlichkeit, hier dann den nächsten Befehl mit der höchsten Wahrscheinlichkeit und so weiter. Und laufe da einfach nach unten durch. Das ist also die spekulative Ausführung. Ich hole mir immer den Befehl, der gerade die höchste Wahrscheinlichkeit hat nächsten befehl bei der gründlichen multipfad ausführung da gehe ich immer beide pfade ab und zwar mache ich das ebenenweise ja das heißt im ersten schritt hole ich mir diesen befehl hier im nächsten beschritt hier verfolge ich den befehl hier auch wenn er 30 prozent wahrscheinlichkeit die wahrscheinlichkeit niedriger ist als dieser befehle hat ja immerhin eine 49 prozentige wahrscheinlichkeit so wenn dann diese ebene abgearbeitet ist dann arbeite ich Ebene ab, also ich ar-

beite gleichmäßig eine Ebene nach der anderen ab, mache also so eine gründliche Multifahrtausführung, wenn man sich das Suchalgorithmen anschaut, dann ist das vergleichbar mit einer Breitenzuche. Das aufwendigste Verfahren ist das Disjoint-Iger-Verfahren, da betrachte ich immer die Restwahrscheinlichkeiten in Summe, also die höchste Wahrscheinlichkeit ist wieder der Schritt 1, das sind hier diese 49% ist immer noch höher als diese 30% in diese Richtung. 34% hier unten ist immer noch höher als diese 30%. 24% wäre jetzt in diese Richtung. Das ist natürlich niedriger als diese 30%. Das heißt, im vierten Schritt wird dann dieser Pfad ausgeführt. Dann wird aus dieser Pfad hier unten ausgeführt mit 24%. Und im sechsten Schritt wird dann dieser ausgeführt. Und im siebten und achten Schritt dann diese beiden. diese drei grundsätzlichen Möglichkeiten. Möglichkeit 1 ist am einfachsten zu realisieren, ist aber möglicherweise nicht besonders gut. Möglichkeit 2 ist auch sehr einfach zu realisieren, braucht aber Ressourcen ohne Ende, exponentielles Wachstum, es werden immer alle Pfade ausgeführt. Und die dritte Möglichkeit ist eben wieder so eine Mischung aus 1 und 2, das Joint-Eager-Verfahren. herzustellen. Was jetzt wieder die beste Möglichkeit ist, hängt letztendlich davon ab, wie viel Hardware will ich spendieren, wie wichtig ist die ganze Geschichte und dann suche ich die für mich beste, passende Möglichkeit aus. Gut, dann sind wir mit der Instruction Fetch Phase zu Ende. wie kann ich die Instruction Fetch Phase optimieren, warum? Weil ich einen Block von Befehlen holen muss und weil jeder Sprung eben diesen Block von Befehlen stört, haben wir sehr viel Zeit für die Sprungvorhersage verwendet. Die nächsten Phasen sind da deutlich einfacher. Nächste Phase ist die Instruction Decode und Rename Phase. Was muss ich da jetzt machen, wenn ich an superskalare Prozessoren denke? Klar, die Intuition oder die Intention ist klar. Befehle zur Verfügung stellen. Das heißt, meine Dekodierungsbandbreite, also die Anzahl der Befehle, die ich zu einem Zeitpunkt dekodiere, die sollte deutlich größer sein als die Zuweisungsbandbreite, weil eben ich dann mehr Auswahlmöglichkeiten habe und einfach Befehle nachweg schmeißen kann. heißt, dass die Bandbreite der Dekodierung ungefähr doppelt so groß sein sollte, wie die Bandbreite der Befehle, die ich an die Execution Units weiterreiche, also wie mein Scheduler praktisch weiterreichen kann. Möglicherweise kann so eine Dekodierung auch mehrstufig sein, also das heißt, Sie wissen vielleicht, warum die ganzen alten Maschinenprogramme, x86 Maschine geschrieben wurde, dass die inzwischen, oder dass die immer noch laufen, dass ich die also nicht neu übersetzen muss, diese alten Programme, auch wenn ich einen i7 Prozessor habe, weil eben intern diese alten Maschinenbefehle, ZISK-Befehle zu RISK-Befehlen umgesetzt werden und sowas passiert zum Beispiel in der Dekodierungsphase. Gut, ich kann auch wieder Hardware spendieren, um so eine Vorkodierung zu machen, also extra Bit spendieren, oder dass ich irgendwelche Umordnungsoperationen mache. Es geht einfach nur darum, dass ich möglichst schnell Instruktionen dem Hardware Scheduler zur Verfügung stelle, damit er die auf die Execution Units draufwirken kann. Natürlich, wenn da irgendwelche Pseudoabhängigkeiten auftauchen, also Write-after-Read oder Write-after-Write-Konflikte, die kann ich in der Regel lösen durch einfache Umwendung. hat, dann stellt die Hardware so quasi versteckte Register, physikalische Register zur Verfügung, die dann eben nur die

Hardware nutzt. Also da kann der Compiler dann nicht zugreifen, sondern die Hardware nutzt dann diese Register, um solche Konflikte aufzulösen. Also zum Beispiel der IA32, also Pensionprozessor, der hat jetzt 8 solcher physikalischen Register und der Pension Pro hat schon 40 solcher physikalischen Register. Also es gibt eben in Anführungszeichen ganz gut verwenden kann. Was natürlich wichtig ist, ich muss wieder eine Hardware bauen, die mir das Ganze realisiert. Das heißt, ich brauche dann irgendwo so eine logische Zuweisung zwischen logischen und physikalischen Registern. was eben in Hardware dynamisch passieren kann. Okay, also Instruction Decode und Rename Phase ist relativ einfach. Ich muss einfach genügend, da müssen genügend Instruktionen rausspringen hier aus dieser Phase in dieses Instruction Window, dass mein Hardware Scheduler, der hier in der Issue Phase angesiedelt ist, dass der eben auf die Execution Units das bringen kann. Was macht jetzt diese Issue-Phase, Befehlszuweisungsphase? Ziel ist einfach, ich habe ein Instruction Window, da stehen zur Ausführung bereite Befehle drin und die sollen jetzt möglichst effizient, schnell, parallel den Execution Units zugewiesen werden. Es gibt die Möglichkeit, direkt vor den Execution Units nochmal eine Reservation Station zu schaffen, dass ich da also so einen kleinen Puffer habe, dass ich da also sich die Instruktionen quasi anstellen können, Jetzt muss nicht sein, kann sein. Es gibt auch die Möglichkeit des sofortigen Dispatches, das heißt die Instruktion wird sofort einer Execution Unit zugewiesen, wenn die frei ist, also ohne Pufferung dazwischen. Das Ganze heißt natürlich Scheduler und der ist verantwortlich für die Auswahl und für die Zuweisung. Nicht ganz trivial, können Sie sich vielleicht vorstellen, Sie haben dann einen Sack voll Befehle, Sie haben einen Sack voll Execution Units und Sie müssen das Ganze schön mappen. Das heißt, da brauchen wir ein bisschen Hardware dafür und die bekannteste Hardware für so ein Verfahren ist das Scoreboard und das Tomasolo-Verfahren und das wollen wir eben auch noch besprechen. Integer-Einheiten, Floating-Point-Einheiten, Multimedia-Einheiten, ja vielleicht auch speziell Masken-Einheiten für irgendwelche Maskierungsoperationen. Also da kann man sich beliebige Sachen vorstellen, was eine Execution Unit alles spezialisiert möglichst gut machen kann. Und die werden dann eben entsprechend mit Befehlen versorgt und führen die aus und liefern das Ergebnis dann in der nächsten Phase, in der Retire-and-Ride-Back-Phase ab. units manchmal auch viele pro operationsart also keine ahnung 45 integer units 34 floating point units je nachdem auf was mein prozessor zielt soll ein möglichst general purpose sein dann versuche ich dann eine schöne mischung zu finden oder ist es ein prozessor der besonders gut multimedia kann dann spendiere ich halt ein bisschen mehr multimedia und so weiter also multimedia einheit und zwar was gibt so als bekannte einheiten einfache integer einheiten so wie wir sie besprochen haben bis jetzt, dann kann man auch Multiplikation und Division optimieren, indem man ein bisschen komplexere Einheiten dafür spendiert, aber Integer, Multiplikation und Division. Da gibt es natürlich die entsprechenden Operationen in Fließpunkten, also in Floating Point, meistens nach dem IEEE-Standard. Dann spezielle Load-Store-Einheiten, die also die Adressrechnung praktisch beschleunigen, Datacache zugreifen kann und dann natürlich Multimedia, großes Thema heutzutage, dass ich eben möglichst schnell alle Operationen, die mit Grafik zu

tun haben, möglichst schnell ausführen kann. Ja, da ist ja oft so, dass so eine SIMD-Operation, also die gleiche Operation auf allen Pixel, dass das sehr oft auftaucht, wenn ich Masken filter, wenn ich irgend so was mache. Also liegt es natürlich nahe, dass ich spezielle Execution Units baue, die das ganz gut können. Gut, wenn dann also in dieser Phase die Befehle parallel natürlich alle bearbeitet werden, dann liefern die das Ergebnis zurück. Und jetzt haben wir normalerweise hätten wir beim Datenpfad hier eine einfache Write-Back-Phase, also das Ergebnis einfach wegschreiben, entweder ins Register-File oder in den Speicher, wenn ich eine Store-Operation habe. ist es hier erweitert um Retire and Ride Back. Warum? Haben wir jetzt ja vorher darüber gesprochen. Möglicherweise liegen da jetzt Ergebnisse vor, die ich gar nicht brauchen kann, weil ich die spekulativ ausgeführt habe und die ich dann einfach wegschmeißen muss. Das heißt, die Ride Back Phase ist nicht mehr so ganz einfach, sondern ich muss da ein bisschen mehr machen. Also das Ziel ist natürlich die endgültige Übernahme des Ergebnisses, wenn das berechnet ist. Kann natürlich so ein Rollback auftreten, wenn was falsch berechnet wurde, muss ich auf Seiteneffekte achten, dass ich alles ordnungsgemäß zurückdrehe, damit keine fehlerhaften Zwischenergebnisse übrig bleiben. Und natürlich brauche ich diese Phase auch, wenn irgendwelche Interrupts kommen. Interrupts kommen dann gleich auf der nächsten Folie. Ich habe so einen Puffer mit Befehlen, die anstehen zur endgültigen Erledigung. Und ich habe zwei Möglichkeiten. Entweder ich übernehme das Ergebnis und speichere es dann weg im Registerfall oder im Speicher. Oder ich verwerfe das Ergebnis. Also mit oder ohne Commitment. Dementsprechend gibt es drei mögliche Aktionen in dieser Phase. Eine Aktion ist Completion. Das heißt, das Ergebnis ist im Pufferregister jetzt verfügbar. Dann gibt es Commitment. Beim Commitment wird das Ergebnis endgültig übernommen. Also da kann ich dann auch kein Rollback mehr fahren, das ist dann vorbei. Das steht dann im Speicher oder im Register drin. Und es wird natürlich danach, nachdem es Committed ist, aus dem Retirement Buffer, also aus dem Puffer entfernt. Und da gibt es ein einfaches Removement. Das heißt, ich weiß schon, dass ich diesen Befehl auf keinen Fall brauche, weil er spekulativ falsch ausgeführt wurde. gültig erklären und entferne den aus dem Puffer. Ja, wir haben gerade von Precise Interrupt gesprochen. Wenn ich eine sequenzielle, streng sequenzielle Maschine habe, dann ist so ein Interrupt natürlich einfach. Ich unterbreche die Maschine und zwar unterbreche ich zwischen zwei Befehlen. Der eine Befehl ist fertig ausgeführt, der andere ist noch nicht angefangen. wo dieser Interrupt zuschlägt. Wenn ich jetzt so eine komplexe Hardware habe, mit Superskalarität, mit paralleler Verarbeitung, dann muss ich natürlich schauen, dass dieser Interrupt precise wird. Das heißt, ich brauche einen wohl definierten, gesicherten Prozesszustand. Und dafür gibt es natürlich Bedingungen. Es ist wie im sequentiellen Fall. Alle Befehle vor dem Interrupt, die wurden vollständig ausgeführt. dann darf die Befehle, die nach dem Interrupt kommen, darf natürlich nicht ausgeführt werden. Und die dürfen vor allen Dingen keinen Einfluss auf den Professorzustand haben. Das heißt, eigentlich müsste ich beim Interrupt meine schöne Datenfahrt, die schöne Befehlspipeline anhalten, diese Pipeline leer laufen lassen, damit ich so einen wohl definierten Zeitpunkt habe. Und genau so muss die Hardware auch gebaut

sein. Gut, ist glaube ich klar, dass das im sequentiellen Fall relativ einfach ist, hier bei so einer superskalaren Architektur, dass ich da wieder Zusatz-Hardware brauche, die eben dafür sorgt, dass es einen wohl definierten Zeitpunkt gibt, also alle Befehle vor noch vollständig ausgeführt werden. sagen in den Befehlspipeline. Okay, gut, das waren also jetzt die fünf Phasen, Instruction Fetch, Instruction Decode, Issue, Execute, Write Back und Retirement und wir haben ja unterwegs darüber gesprochen, dass diese dynamische Kontrolle dieser vielen Ressourcen eben aufwendig ist und Hardware erfordert und dafür wollen wir jetzt Ideen und Verfahren nochmal ein bisschen genauer anschauen. und es gibt die Idee der entkoppelten Architektur, die der Herr Thomas Huller entwickelt hat. Fangen wir an mit dem Scoreboard. Ich muss dynamisch Konflikte checken, in Hardware, in Laufzeit, in Real-Time. Ich muss natürlich den gesamten Prozessorzustand kennen, das heißt, ich muss wissen, welche Ressourcen alle in Arbeit sind, wer welche Ressourcen belegt, das heißt, Befehle bearbeite ich gerade, brauche ich also einen Befehlszustand. Dafür brauche ich eine Tabelle für alle Befehle, die aktuell bearbeitet werden. Ich muss eine Übersicht haben, was die Execution Units gerade treiben. Das heißt, ich brauche eine Tabelle, wo drin steht, was die Execution Units gerade treiben. Und ich muss meine Ergebnisse ja irgendwann wegspeichern. Ich brauche also meine Zielregister, ob die verfügbar sind, in welchem Zustand sie sind. Das heißt, ich brauche die dritte Tabelle für den Registergebniszustand. Scoreboard, der speichernde Teil, eine Tabelle für die Befehlszustände, eine Tabelle für die Execution Units und eine Tabelle für die Registerergebniszustände. Das Scoreboard sammelt jetzt die ganzen Informationen und anhand dieser Informationen trifft eben der Scoreboard Manager seine Entscheidungen, ob er jetzt ein Assignment durchführt, ob er ein Issue durchführt oder ob er Stalls auslöst, also Wartezüge. Ich habe hier mein Ergebnisregister, ich habe ein Source-Register, ich habe hier meine Execution-Units, ich habe hier den Manager, der über diese Scoreboard-Datenstruktur seine Informationen enthält und der das Ganze dann steuert. Wir gehen jetzt mal von der Pipeline aus, die wir die ganze Zeit besprochen haben, write back. Wir wollen jetzt Memory, also Store-Operationen überhaupt nicht betrachten, sondern alles außer Store-Operationen. Außer Load-Store-Operationen interessieren wir uns jetzt gerade mal nicht. Wir wollen einfach diesen Register zu Registerfluss noch betrachten. Macht die Sache ein bisschen einfacher. Das Scoreboard ist eh schon kompliziert genug. Klar, haben wir ja vorhin schon gesehen, diese zwei Schritte haben wir bis jetzt immer als Instruction Decode bezeichnet. Ist halt ein bisschen aufwendiger und deshalb in zwei Phasen unterteilt, in Instruction Issue und in Instruction Schedule. So, jetzt müssen wir uns überlegen, in den einzelnen Phasen, was muss das Scoreboard alles machen. Also in der Instruction Fetch Phase, Befehlshaus Phase, hat das Scoreboard erstmal gar nichts zu tun. Befehl Zustandstabelle eingetragen werden. Dann muss ich checken, ich weiß ja jetzt, welche Ressourcen der Befehl braucht. Ist die Execution Unit verfügbar? Das heißt, ich muss in die Tabelle schauen, wo die Execution Units gelistet sind, ob die frei ist. Dann muss ich schauen, ob mein Zielregister frei ist, ob irgendein Right-after-Right-Konflikt möglicherweise vorliegt, dass ich irgendeine Operation überhole und dann Right-after-Right habe. Das heißt, ich muss nachschauen,

der freies also registergebnis zustand checken wenn es frei ist dann hand drauf legen reservieren für für den befehl der gerade dann ausgeführt wird und wenn nicht dann habe ich einen konflikt ja dann muss ich es tollen ja und muss erst mal dafür sorgen dass von hinten nichts überläuft das heißt ins rock schon fertig erstmal stoppen und wenn ich dann keinen konflikt habe dann sage ich Nächste Phase wird eingeleitet, nämlich die Instruktion Schedule Phase. In der Schedule Phase wird erstmal der Execution Unit Zustand abgedatet. Das heißt, da steht jetzt drin, welcher Befehl ausgeführt wird, was ist das Target, also das Ziel, was ist Source Register 1, was ist Source Register 2. Dann mache ich Checks auf Konflikte, also Read After Write Check. dann habe ich ein Problem, dann muss ich FNVL umbenennen oder was anderes machen, also ich muss im Registergebniszustand nachschauen, ob da was ist, wenn ja, dann muss ich die Quelle als nicht verfügbar markieren und dann entsprechend ein Stall machen, bis diese Quelle verfügbar ist. Schedule als erledigt betrachten und gehe jetzt in die nächste Phase, in die Execution-Unit-Phase. So, Execution-Unit-Phase ist einfach, da wird nur ausgeführt und wenn die Ausführung fertig ist, dann wird es entsprechend markiert, wenn ich also fertig bin und Mem-Phase wollen wir nicht betrachten, also wir wollen keine Lot-Stop-Effille betrachten. Write-Back-Phase, da muss ich jetzt Das heißt, ist das Zielregister gleich dem Quellenregister eines vorherigen Befehls? Wenn ja, muss ich warten, bis es frei ist, muss also ein Stall einfügen. Und wenn kein Konflikt ist, dann kann ich mein Ergebnis ins Zielregister schreiben und den ganzen Befehl als erledigt betrachten. In dem Moment wird er natürlich auch aus dem Scoreboard entfernt. Wir haben also hier so ein Beispiel für eine Scoreboard-Aktion. Das ist der erste Schritt, das sind jetzt immer so zeitliche Aufnahmen. Hier 1, 2, 3, 4, 5, 6 Befehle, die wir betrachten wollen. Und so ein X in der Tabelle bedeutet, dass der jeweilige Schritt abgeschlossen ist. Das heißt, der sechste Befehl ADEF, der wurde noch gar nicht begonnen. Also das steht noch nicht mal im Instruction Issue. Die 5, 4 sind alle in der Instruction Issue Phase. Die zwei hier oben sind schon gescheduled. Die sind auch schon ausgeführt und der erste Load Befehl ist auch schon in der Write Back Phase. So, hier unten in der nächsten Tabelle, in der Execution Unit Zustandstabelle, da steht natürlich drin, mit was die Execution Units beschäftigt sind. Und da sehen wir, dass praktisch die Integer-Phase momentan mit dem Load-Befehl beschäftigt ist, nämlich mit diesem Load-Befehl hier. die Multiplikationsunit für den Mult-F-Befehl, der hier auftaucht, und die zwei Integer-Units hier, also die Add-Unit, Entschuldigung, die sind keine Integer-Units, die sind Floats, die Add-Unit hier ist für diesen Sub-F-Befehl vorreserviert, und die Divisionsunit ist für den Div-F-Befehl vorreserviert. Ja, die sind also vorreserviert, werden als bisher gekennzeichnet, und es wird natürlich mitgeführt, was ist das Ergebnisregister, also das Zielregister des geplante, Und was sind die Source-Register? Und in Klammern steht, dass die eben noch nicht zur Verfügung stehen, dass die aus Schritten von vorher kommen. Und hier unten sind quasi die Ergebnis-Register, genauso wie sie hier bei den Befehlen auftauchen. Welcher Befehl, aus welcher Execution-Unit kommt praktisch das Ergebnis, was dann ins entsprechende Register geschrieben wird. So, jetzt takten wir einfach ein bisschen weiter. Ja, und wir haben ja schon festgestellt, also

bevor wir jetzt weiter takten, haben wir eben festgestellt, dass die Mult-F im Write-Back-Phase ist, die F ist in Write-Back-Phase und hier sind entsprechend die Ausführungszeiten, die wir haben. Und schauen mal auf Konflikte. Und dann sehen wir hier, dass hier das Register F0 geschrieben wird und hier das Register F0 gelesen wird. Das heißt, wir haben hier einen Read-After-Write-Konflikt, der zu lösen ist. Das heißt, diese Division darf erst dann gestartet werden, wenn diese Multiplikation beendet ist. Und das muss der Scoreboard natürlich erkennen. Und hier haben wir einen Write-After-Read-Konflikt. f6 lesend verwendet und die haben möglicherweise eine schnelle addition ja und ja da müssen wir natürlich auch aufpassen dass ich ersten right back mache auf dieses register f6 wenn es hier schon gelesen wurde also so ein right after read konflikte auch aufgelöst werden das kennt natürlich Erbelegung wieder. Das sind also die entsprechenden Konflikte erkannt und wird entsprechend durch Stalls erledigt. Wir haben also zwei Konflikte jetzt drin, die wir auflösen müssen und ich weiß genau, wie viele Taktzyklen meine Execution Units brauchen und kann da relativ einfach die entsprechenden Stalls ausrechnen, die ich warten muss, um diese Konflikte zu Und wenn ich dann diese zwei Konflikte gelöst habe, dann kann auch die letzte Operation, ja das wird hier die letzte Operation sein, die ausgeführt wird, weil die musste warten, dass hier alles korrekt geschrieben wurde. Und hat dann nach der Konfliktlösung, wird dann diese Divisionsoperation nochmal ausgeführt. Überblick erhalte ich dadurch, dass ich das Ganze in Tabellen baue und da meine Konflikte ganz gut eben ermitteln kann und dadurch die korrekte Reihenfolge sicherstelle und auch so wenig wie möglich Stalls in diese ganze Operation mit einbringen. Einfach mal das Beispiel durchspielen, dann wird es klar. Okay, also Scoreboarding ist eine sehr gute Methode, die man auch einfach in Eine ähnliche Idee ist schon sehr viel früher aufgekommen und zwar ist es die sogenannte entkoppelte Architektur. Was ist eigentlich das Problem? Das Problem ist in jeder Pipeline, dass ich einen Befehl nicht verarbeiten kann, weil meine Quelldaten eben nicht zur Verfügung stehen. Das ist ein echter Konflikt. Wenn ich keine Quelldaten habe, dann kann ich auch keine Operation darauf ausführen. Unechter Konflikt ist, wenn ich nicht genügend Register habe. Wenn ich also ein Register nicht beschreiben kann, weil das Register durch irgendeinen anderen Befehl noch belegt ist. Dann nehme ich halt ein anderes Register, also Umbenennungsstichwort. Solche Konflikte kann ich in der Regel lösen. Das heißt, unechte Konflikte kann ich verhindern, indem ich eine geschicktere Registerzuordnung mache. Also das kann dann auch der Compiler schon machen. Das kann ich statisch machen. Idee dazu, ich verwende einfach überhaupt keine Register. Dann kann ich auch keine Konflikte auf Register haben. Das heißt, ich baue eine ganz andere Architektur und wenn überhaupt, dann habe ich so ein kurzes Pufferregister, also Scratchregister und wenn man diese Idee jetzt zu Ende denkt, das heißt, dass die Quelldaten die Ausführung eines Befehls triggern, dann bin ich eigentlich bei einer Datenfluss Idee. Also die nicht mehr die Befehle fließen, das sind wir einem anderen abarbeiten, sondern die Daten fließen und die Daten triggern die Ausführung von Befehlen. Also immer dann, wenn meine Quelldaten zur Verfügung stehen, führe ich diesen Befehl aus. Das ist eine ganz andere Idee, eine ganz andere Architektur wäre das. Die grund-

sätzliche Idee dieses Datenflussrechners ist hier nochmal in der Skizze gezeigt. wir vielleicht auch als Maschinenbefehl bezeichnen, ich habe da irgendwo drin stehen, eine Operation, die auszuführen ist auf zwei Quelldaten. Also hier ist dann als Operation, Addition, Subtraktion, Multiplikation, logische Operation, irgendwas und hier stehen meine zwei Quelldaten. Dann habe ich irgendwo einen Speicher, dieser Speicher beinhaltet diese ganzen Instruktionszellen Das kann man sich noch wie herkömmlich vorstellen. Jeder Mikroprozessor-Chip besteht natürlich aus einem Prozessor und einem Speicher, also irgendeinem Cache. Angeschlossen daran ist natürlich noch ein großer Cache. Aber jetzt ist es natürlich nicht so, dass die Befehle zur Ausführung kommen, sondern hier fließen die Datenpakete, die im Wesentlichen die Quelloperanten beinhalten. Und diese Quelloperanten werden dann auf diese Instruktionszellen verteilt. Und in dem Moment, wo eine Instruktionszelle vollständig ist, vorhanden sind, werden diese Instruktionszellen zur Bearbeitung zum Prozessor gebracht und die Operation wird ausgeführt. Also das ist die grundsätzliche Idee von so einem Datenflussrechner. Also sobald alle Quellenoperanten verfügbar sind, das ist quasi der Trigger für die Ausführung, dann stehen die zur Ausführung an und dann können die jetzt einfach abgeholt und ausgeführt werden. wird, sondern die Verfügbarkeit der Quelldaten bestimmt die Ausführungsreihenfolge meines Programms. Ja, diese Idee wurde schon sehr früh eigentlich mal versucht umzusetzen. Da gab es eine ganz bekannte IBM-Maschine, 36091 mit Floating Point Unit, also sehr alt, Apollo-Mission, der NASA begleitet natürlich, also es war für die damalige Zeit einer der leistungsstärksten Rechner. Und da ist ansatzweise so ein bisschen die Idee von dem Datenfluss. Allerdings, da es nur ansatzweise ist, hat man es dann doch eher als entkoppelte Architektur bezeichnet. Aber es geht schon die Grundidee des Datenflussrechners ein. Also wir haben auch Reservierungsstationen und diese Reservierungsstationen könnte man jetzt vergleichen Befehlszellen der Datenflussarchitektur und die Registernamen tauchen nicht explizit auf, sondern im Wesentlichen die Datenflussbeziehungen. Das heißt, ich erreiche die Daten dann gleich weiter, ich berechne also ein Ergebnis und ein Ergebnis wird ja in der Regel vom nächsten Befehl dann benötigt und dieses Ergebnis wird dann als Quelle an den nächsten Befehl praktisch weitergeleitet. Das heißt, ich habe also hier in der herkömmlichen Weise meine Befehle, Ergebnisregister, also Targetregister mit zwei Quellregistern und das wird praktisch im Wesentlichen ersetzt durch die Pfeile, das heißt, dieses Ergebnis, was ich hier berechne durch die Load-Operation, das taucht hier nochmal auf und taucht hier nochmal auf, also gebe ich das dann quasi an diese Befehle gleich weiter und brauche dann nicht explizit diese Register verwenden. Also Register werden quasi ersetzt durch Datenflussbeziehungen. Aufbau ist dann eben so eine Architektur wo diese Daten durchaus fließen. Das heißt also, dass ich hier meine Execution Units habe, Load Unit, Multiplikations Unit, Additions Unit und ich habe jetzt hier keine expliziten Registernamen, sondern ich habe solche Beziehungen durch diese liefert bei dieser Additionsunit und das Ergebnis hier abliefert bei der Multiplikationsunit. Diese Multiplikationsunit wiederum ihr Ergebnis abliefert hier bei dieser Additionsunit. Also ich habe eben diese Pfeilbeziehungen, die quasi den Datenfluss darstellen und sobald vorne Execution Unit beide Quelloperanten

verfügbar sind, triggert die und liefert ihr Ergebnis dann wieder ab, die dieses Ergebnis dann ausführen wollen. Es ist also eine gute Idee eigentlich, ich lasse nicht Befehle fließen, sondern lasse Daten fließen, allerdings ist es sehr schwer, das Ganze in Harge abzubilden. Und das ist wahrscheinlich auch der Grund, warum der Datenflussgedanke als theoretisches Modell in der Informatik sehr beliebt ist kenne ich zumindest nicht. Das Problem ist nämlich der Transport des Ergebnisses zu den Reservierungsstationen. Ich brauche da ja also erstmal ein Netzwerk mit voller Konnektivität. Das heißt, ich muss von jeder Execution Unit jede andere gut erreichen. Ich muss genügend Execution Units haben. Das Ganze muss natürlich sehr schnell sein, weil das ist ja der Knackpunkt, damit es nicht zum Engpass wird. Und es gab Ansatz von IBM, da gab es den sogenannten CDB, Common Data Bus, aber dieser Common Data Bus war eben nicht schnell genug, also der hat die ganze Maschine ausgebremst, wenn man so will, und wir sprechen vom Jahr 1967, damals waren die Busse natürlich auch entsprechend langsam. Also man hat es probiert, es hat auch einigermaßen funktioniert, aber die herkömmlichen Architekturen scheinen eben besser zu funktionieren. Ja, wenn man jetzt so eine Tomasolo, also so eine inkopelte Architektur hat, wie kann das denn dann überhaupt funktionieren? Also wie organisiere ich diesen Datenfluss? Wie organisiere ich das, dass die Ergebnisse gleich als Quellregister an die Antearchitektur weitergegeben werden? Da müssen wir wieder die entsprechenden Phasen betrachten. Also ich habe die Issue-Phase, ich weise den Befehl einer freien Execution Unit zu und dann hole ich meinen Quelloperanten. Register oder eben aus dem entsprechenden Ladepuffer oder er kommt von einer anderen Execution Unit. Das heißt, bei der Ausführung hole ich die Operanten von den zugehörigen Execution Units, also die fehlenden Operanten, ich löse die Konflikte und ich muss natürlich warten, bis mein Operant verfügbar ist, dafür beobachte ich den Common Data Bus. Und wenn dann meine Operanten alle da sind, meine Operation ausgeführt wird, dann schreibe natürlich in den Common Data Pass und davon aus dem Common Data Pass holen das alle anderen, die gerade auf dieses Ergebnis warten, um dann ihre Operation auszuführen. Auf diese Art und Weise, da ich ja den Datenfluss festlege, können überhaupt keine Right-after-Read- und Right-after-Write-Konflikte auftreten. Also die brauche ich nicht beobachten. Die Architektur an sich löst diese Probleme. Das kann in dieser Architektur nicht passieren, brauche ich also auch weiter nicht beachten. dass ich diese Konflikte gar nicht betrachten muss. Ja, letztendlich, wenn man das jetzt umsetzen will, dann ist es am einfachsten, wenn man das genauso macht wie mit dem Scoreboard. Ich brauche wieder alle Informationen, ich brauche die Befehlszustände, ich brauche die Zustände der Execution Units und ich brauche die Zielregisterzustände. wieder so einen Scoreboard-Ansatz fahren. Wir haben also die Befehlszustände genau wie vorher. Wir haben die gleichen Befehle als Beispiel wie vorher genommen. Und wir tragen also diese Befehle erstmal wieder ganz normal in unsere Befehlzustandstabelle ein. Bei der Execution-Units-Tabelle, die sieht so ähnlich aus. Wir haben also auch hier wieder unsere Execution-Units. Da steht drin, welchen Befehl die ausführen können, ob die jetzt belegt sind schon oder nicht. drin entweder wenn das aus operant verfügbar ist dann woher gekommen ist oder in klammern steht drin von welcher execution

unit erwarte ich denn jetzt mal ins aus operanten also das heißt diese execution unit wird praktisch befeuert von 1 und von low 2 um dann irgendwie viel aus rechnen, Execution Unit. Und das Register Ergebnissustand, da steht jetzt drin, von welcher Execution Unit ein Ergebnis abgeliefert wird, also Register F0 bekommt sein Ergebnis aus der MULT1 Unit, F2 aus der Load2 Unit, F6 aus der Add2 Unit, Add1 aus der F8 Unit und F10 aus der MULT2 Unit. Das heißt also, was den Execution Unit Zustand dann steht er unter S1 und da steht drin, woher er gekommen ist. Oder er steht in Magenta-Farben, in Klammern und da steht drin, von welcher Execution Unit dieser Operant abgeliefert wird. Gut, also man sieht, man kann das Scoreboard auch ein bisschen missbrauchen, um diese Thomas-Solo-Algorithmus zumindest abzubilden und das dann als Vorlage zu nehmen, um daraus dann eine entsprechende Hardware zu erstellen. Gut, der Vorteil ist eben, dadurch, dass ich auf Ergebnisse sowieso warte, können keine Write-after-Read und Write-after-Write-Konflikte auftreten, weil eben der Befehl div f hier zum Beispiel, div v zum Beispiel sowieso so lange wartet, bis dieser Eingabeoperand f0 vorliegt. Erstmal muss diese Multiplikation hier passieren, das Ergebnis f0 liefern und dann wird erst dieser Befehl getriggert. Das heißt, dieses Warten geschieht ganz automatisch, architekturbedingt. Und von daher gesehen, brauche ich solche Konflikte auch nicht betrachten. Okay, wir liegen sehr gut in der Zeit. Hier ist nochmal die letzte Folie, da ist es einfach nochmal gezeigt, wann die letzte Operation hier ausgeführt wird. Und dass es eben keine Konflikte gibt. ja, vielleicht zum Abschluss dieses Kapitels Superskalarität noch ein schlauer Spruch vom Ludwig Mies van der Rohe, Architecture starts when you carefully put two bricks together, there it begins. Ich hoffe Ihnen mit diesem großen Kapitel Superskalarität gezeigt zu haben, dass eine parallele Architektur, wo mehrere Execution Units und mehrere Befehle gleichzeitig in Bearbeitung sich befinden, sehr, sehr sorgfältig aufbauen muss, weil alles hängt von allem anderen ab. Ja, ich kann sehr leicht Dominoeffekte produzieren, ich kann sehr leicht Ressourcen verschwenden, ich kann sehr leicht ineffizient arbeiten. Also man muss eben sehr, sehr, sehr genau darauf achten, was baut man wie zusammen. Man muss immer Datenabhängigkeiten betrachten, man muss Kontrollflussabhängigkeiten betrachten und man muss eigentlich bei jeder kleinen Architekturentscheidung jedes Mal viele Simulationen fahren, Ja, spendiere ich ein Bit mehr oder weniger, spendiere ich eine Verbindungsleitung mehr oder weniger, spendiere ich eine logische Einheit mehr oder weniger. Alles muss zusammenpassen. So eine Architektur ist wie eine Mannschaft, also jeder muss mit jedem anderen interagieren, mit jedem anderen Teil der Architektur und alles muss sehr gut aufeinander abgestimmt sein und dann kann ich Leistung rausholen und Performance ist sehr ganz wichtig. auf Mikroarchitekturebene. Wir wollen uns dann ab nächsten Freitag beschäftigen mit großen Parallelrechnern. Das heißt, wir wollen ganze Prozessoren miteinander verbinden, mit schnellen Netzwerken, um dann eben noch mehr Leistung rauszuholen.