wir haben das letzte mal aufgehört im thema super skalarität und wir haben uns zum schluss angeschaut den power pc das ist eine mikroprozessor architektur von ibm die und Motorola entwickelt wurde. Ziel war von Anfang an einen RISC Prozessor zu entwickeln, der 64 Bit hat. Es gab auch 32 Bit Varianten, aber Entwicklungsziel war von Anfang an 64 Bit. So, wir bewegen uns ungefähr im Jahr 2005 und diese Architektur war natürlich zielführend oder das war ein Vorbrechel praktisch in Richtung einer modernen, superskalaren schaltbild anschauen dann kommt ihnen natürlich einiges sehr bekannt vor also wir haben hier eine harvard architektur das heißt hier oben steht der instruction cache hier unten steht der data cache also getrennte speicher für befehle und für daten heißt harvard und dann haben wir hier rot gezeichnet den normalen befehlsfluss das heißt wir haben ganz grob wieder die phasen instruction fetch instruction decode danach kommt execute writeback und memory Back-Phase. Was denn das Execute-Phase betrifft, da haben wir hier unten sehr viele Execution-Units, die da parallel liegen und die eben mit Befehl versorgt werden müssen und dann läuft es hier unten, das ist eine Register-Register-Maschine, läuft es hier unten die Execute-Phase in den Registern, das ist halbwegs problemlos. Das große Problem ist, diese Architektur genügend Befehle gleichzeitig zuzufügen, wir haben hier eine Parallelverarbeitung und Phase Instruction Fetch und Instruction Decode besonders wichtig und dementsprechend auch ein bisschen deutlich mit Hardware aufgepolstert. Also man sieht es mit Sachen wie BHT steht für Branch History Table, BTAG für Branch Target Address Cache und für RAS, also Return Address Tag, was das alles bedeutet. Darauf werden wir im Folgenden eingehen. Also heute die Vorlesung und auch nächste Woche Dienstag die Vorlesung. das Entscheidende und im Moment müssen wir uns erstmal merken, okay, wir brauchen das, um ausreichend schnell Befehle zu unseren Execution Units zu bringen. Hier ist noch ein bisschen im Detail erklärt, was es ist. Also hier steht Load-Star-Architektur, Befehlslänge 32-Bit. Das war die ersten Implementierungen, aber wie gesagt, die Architektur hat eigentlich von Anfang an auf eine 64-Bit-Architektur gezielt. Es gibt natürlich Load-Star-Einheiten, um Daten zwischen Registern und Data Cache zu übertragen. Das ist dann da unten. Die Daten müssen ja zur Not in den Speicher zurückgeschrieben werden. Hauptsächlich haben wir Register-Register-Befehle, aber es gibt natürlich auch Load-Store-Befehle, die ausgeführt werden müssen. Wir haben Sprungvorhersage implementiert, erstmalig hardware-technisch. Und das resultiert natürlich in Zusatz-Hardware. Und für diese Sprungvorhersage brauchen wir die Branch-Einheit. Diese Branch-Einheit besteht im Wesentlichen aus Sprungziel-Adress-Cache, also B-Tag, aus Branch-History-Table und Return-Adress-Tag. Das sind die Speicherelemente, die ich hier brauche. Und natürlich brauche ich dann auch noch eine Verknüpfung zwischen diesen Speicherelementen. Und dafür ist praktisch dieser Teil der Hardware hier im Wesentlichen vorgesehen. nächsten Kapitel. Wir müssen natürlich relativ schnell dann auch die Befehle, die wir haben, dekodieren und schreiben die dann in den Befehlspuffer und dann arbeitet hier ein Hardware Scheduler, der eben diese Befehle dann auf die Execution Units verteilt. Das heißt, dieser Scheduler, der hier erwähnt ist, das ist ein Hardware Scheduler, der da arbeitet. Die Execution Units, das ist relativ Standard, also

wir haben Integer-Einheiten, oder als Pipeline, möglichst kurze Pipeline hier, also eine Latenz von drei Taktzyklen vorgesehen. Es gibt Floating Point Einheiten, man verendet natürlich IEEE Standard Floating Point Darstellung, da haben wir ja in Grundlagen Rechner Architektur auch schon drüber gesprochen. Und es gibt dann auch, man hat erkannt, dass Multimedia auf modernen Rechnern eben sehr wichtig ist, dementsprechend entsprechende ALUs, die Multimedia besonders gut können, Umordnung, Konvertierung, dass das alles möglichst schnell passiert, wird das quasi die Algorithmen hierfür in Hardware gegossen. Dann gibt es so eine Retirement-Einheit, da ich parallel und superskalar arbeite, kann es passieren, dass hier in diesem Teil sich Instruktionen überholen und diese möglicherweise dann falsche Reihenfolge muss ich natürlich wieder richtigstellen. Das heißt, das brauche ich, wenn ich normal sequenziell arbeite, nicht. Wenn ich parallel arbeite, ist das wieder Zusatz-Hardware. für diese Retirement Einheit. So, Register klar, die sind dann den Execution Units zugeordnet, Register, Register Architektur und wir haben zwei MMUs, eine MMU für den Zugriff auf die Befehle, die ist da oben angesiedelt zwischen Instruction Cache und der Befehlsverarbeitung und eine MMU, die ist hier unten angesiedelt, das ist dann für den Daten Cache, also das ist dann für die Loadstore Befehle praktisch. für eine superskalare Architektur. Was bedeutet jetzt genau Superskalarität? Dieses ganze Kapitel hat jetzt mehr oder weniger so gute Beispiele gezeigt für Superskalarität. Aber wenn wir das jetzt einfach mal in Definitionen fassen wollen, was gehört alles zu Superskalarität? Superskalarität heißt einmal, dass ich mehr als einen Befehl in einem Instruction-Fetch-Zyklus hole. Das heißt, pro Taktzyklus hole ich, keine Ahnung, 2, 4, 6, 8 Befehle, also mehrere Befehle gleichzeitig. und deshalb spricht man auch von Instruction Level Parallelismus, weil diese Befehle werden gleichzeitig, also parallel abgearbeitet. Wir gehen momentan davon aus, dass die Eingabeanweisung Stromlinie heißt, das heißt, ein Befehl nach dem anderen wird geholt und es wird auch so abgebildet, dass die Linie abgearbeitet wird. werden, das heißt von oben nach unten, auch wenn sie in der Ausführung parallel verarbeitet werden. Wir haben einen Hardware Scheduler, auch der gehört zur Superskalarität und der sorgt für eine dynamische Zuweisung, das heißt hier kann diese Linearität in der Befehlsausführung mal unterbrochen werden. Befehle können sich möglicherweise in der X-Phase überholen, aber ich sorge dafür, dass ich das eben am Ende wieder richtig stelle durch die Retirement Unit. zugewiesenen Befehlen, die ändert sich auch dynamisch. Dynamisch ist immer während der Laufzeit und da ich einen Hardware Scheduler habe, der schaut halt immer, okay, habe ich einen Befehl, habe ich eine Execution Unit, die passt und frei ist, dann schmeiße ich diesen Befehl auf diese Execution Unit und da ich eine gewisse Anzahl von Execution Units habe, die nicht immer ausgelastet ist, kann es natürlich sein, dass diese Anzahl an zugewiesenen Befehlen sich während der Laufzeit ständig ändert. Eine Grundregel ist, dass ich mehr um das Ding einigermaßen zu beschäftigen, als man die Zuweisungswandbreite des Hardware-Schedulers beträgt. Ich weiß ja immer nicht, habe ich jetzt lauter Instruktionen, die ich gleichzeitig bearbeiten will, die alle mit der Integer-Unit arbeiten wollen oder habe ich gemischt Integer- und Floating-Point-Befehle und da ich diese Mischung eben nicht kenne statisch, versuche ich halt so eine

Abschätzung zu machen mit wieder Simulationen, Lastuntersuchungen, wie sind die typischen Situationen, die auftreten werden okay, ich baue jetzt vier Integer-Units, zwei Floating-Point-Units, drei Multimedia-Units. Ja, es ist natürlich hier ein bisschen für einen General-Purpose-Prozessor schwierig, da dieses genaue Ding einzustellen. Manchmal habe ich halt mehr Multimedia-Aufgaben, manchmal habe ich mehr wissenschaftliche Simulationen und der Unterschied im Profil ist natürlich enorm. Also ich versuche hier, das Ganze möglichst gut aufzubauen. Es wird kein generelles Optimum geben. sondern die halt besonders gut dann Multimedia können, die ich aber vielleicht nicht so gut brauchen kann für andere Sachen. Diese Superskalarität, die berührt mich als Programmierer erstmal nicht, weil alles auf Hardware-Ebene passiert. Das heißt, erkennbar ist sie eigentlich nur, wenn ich ins Mikroarchitekturlevel schaue, also wenn ich wirklich tief in die Hardware reinschaue. Und das ist für mich natürlich ein Vorteil, weil als Programmierer habe ich ein einfaches, seriöses Programmiermodell. Das heißt, ich brauche mich da gar nicht darauf einstellen, mit in Berührung kommt, ist der Compiler, aber ich schreibe ganz normal mein Programm, wie ich das kenne. Das ist natürlich ein sehr großer Vorteil. Und das Prinzip, was zur Performance-Steigerung führt, heißt Instruction-Level-Parallelism, also mit ILP abgekürzt und es bedeutet praktisch, dass ich die feingranulare Parallelität auf der Befehlsebene nutze. Das heißt, dass ich dynamisch in Hardware parallelisiere Und ich konzentriere mich darauf, einzelne Befehle möglichst parallel auszuführen. Gut, ich weiß nicht, wie man sagen will, unter Art Weiterentwicklung von dieser Architektur ist natürlich die VLEW-Architektur, also Very Large Instructional World Architektur. Und auch hier versuche ich mehrere Befehlswörter parallel zu bearbeiten. diese Befehlswörter in einen Maschinenbefehl. Also vorher haben wir einzelne Instruktionen, die wir parallel ausführen und hier habe ich schon, dass der Maschinenbefehl praktisch mehrere Instruktionen zusammenfasst. Dementsprechend breit ist der, also der ist zwischen 128 Bit und 1024 Bit breit. Das heißt, wenn ich jetzt 128 Bit habe und habe eine 32 Bit Architektur, dann packe ich immer vier Befehle in ein Maschinenbefehlswort. statisch, das heißt, der Compiler ist dafür verantwortlich, die richtigen Maschinenbefehlswörter zu produzieren, das heißt, der Compiler sucht nach dem Parallelismus und packt eben unter Abhängigkeit der Datenabhängigkeit mehrere Instruktionen in so ein Maschinenbefehlswort zusammen und dadurch entlaste ich die Hardware ein bisschen, weil im anderen Ansatz war es so, dass die Hardware durch den Scheduler diese Parallelität genutzt hat. Hier mache versucht so viel wie möglich Befehle in einen Maschinenbefehl reinzustopfen und hat dann vorab statisch das Ganze parallelisiert. Dieses Prinzip ist natürlich auch gut geeignet für sogenannte SIMD-Operationen. Also SIM steht für Single Instruction Multiple Data. Zum Beispiel bei Matrix-Operationen habe ich sehr oft den Fall, dass die gleiche Operation für sehr viele Daten angewandt wird. Und wenn ich dann so einen Prozessor in den SIMD-Modus schalte, die parallel auf vielen Daten angewandt wird und kann dadurch auch wieder eine Beschleunigung erzielen. Es ist eine ähnliche Idee, ich versuche Befehle parallel zu verarbeiten, der Unterschied liegt im Detail. Beim PowerPC ist es eben so, also bei dieser Superskalaren Architektur, dass ich die Parallelisierung durch Hardware Scheduler dynamisch

mache. Hier beim Very Long Instruction Word ist es so, dass ich das Ganze eben statisch mache. Durch den Compiler und entsprechend lange Befehlsworte haben. Hier gab es noch eine Frage. Ja, gerne. Und zwar wurde gefragt, ob jetzt der Hardware Scheduler weitere Executions auf einer Linie blockiert, bis der Befehl in die Memphase übergehen kann. Ja, die Frage ist berechtigt und sehr gut. Allerdings, wenn ein Verständnis ist, würde ich diese Frage ganz gern im nächsten Abschnitt behandeln, wirklich detailliert auf Instruction Phase und auf das anschließende Hardware Scheduling ein. Also dann versteht man es vielleicht ein bisschen besser, weil wenn ich das jetzt erkläre, müsste ich vorgreifen. Also ich bitte mir die Frage nochmal zu stellen, wenn wir dann im nächsten Kapitel, das beginnt jetzt nach drei Folien, wenn wir dann so ungefähr bei Folie 20 sind, wo wir dann eben, oder noch ein bisschen später, wo wir dann über ein Hardware Scheduler sprechen. Ist das okay? Also generell die Antwort ist natürlich... Der scheint okay zu sein. Okay, generell die Antwort ist natürlich, der Hardware-Scheduler kann nur dann zuweisen, wenn praktisch eine Execution-Unit sich als frei meldet. Also das ist die Grundvoraussetzung, sonst kann der Hardware-Scheduler nichts zuweisen. Und wenn es dann in der Parallelenverarbeitung Probleme gibt, da ist eine Einheit nach dem Hardware-Scheduler zuständig. Ende des nächsten Kapitels, wo es dann um Tomasulo oder um Scoreboard geht. Also ich muss natürlich Konflikte lösen. In Hardware, in Echtzeit. Also das wird natürlich passieren. Aber der Scheduler, der schaut im Wesentlichen, bis der seine Befehle hat, weiß er, okay, jetzt kann ich diese Befehle zuweisen und dann schaut er im Wesentlichen, ob er eine Execution Unit findet, die momentan bereit ist, diesen Befehl aufzunehmen. Aber im Detail schauen wir uns dann Vorlesung, äh, nicht nächste Vorlesung, sondern nächsten Foliensatz an. Okay, also wir sind bei den Very Large Instruction Words. Die Vorteile dieser Architektur sind, dass ich natürlich ein bisschen schneller takten kann, weil der Hardware Scheduler natürlich eine aufwendige Logik ist und den brauche ich nicht mehr, also der verzögert ja auch ein bisschen, dieser Hardware Scheduler, um in Echtzeit irgendwelche Konflikte zu erkennen und zu lösen. Ich habe dementsprechend Ich habe ja statisch parallelisiert und habe da schon die Datenabhängigkeit berücksichtigt. Und deshalb ist die Zuweisungsbandbreite durchaus höher mit so einer VLEW-Architektur. Die Probleme, die ich mir einhandle mit so einer Architektur, die sind einerseits im Cache, weil ich muss aufpassen, dass alle Daten, die ich brauche, schnell natürlich bei den Execution Units vorhanden sind. Wenn jetzt irgendeine Execution Unit auf ein Datum warten muss, dann verzögere ich ja alle jetzt in Ausführung befindlichen Befehle. Weil ich muss immer, da ich statisch parallelisiert habe, immer auf den langsamsten Befehl warten, wenn ich das alles in einem bestimmten Befehl vor Ort packe. Das heißt, so ein Cache-Fehler wird richtig teuer. Der bezieht sich nämlich nicht auf eine Instruktion, sondern der bezieht sich auf alle derzeit in Bearbeitung befindlichen Instruktionen. ein bisschen darauf achten, dass alle Operationen möglichst dieselbe Ausführungszeit haben. Deshalb ist so ein SIMD-Modus natürlich toll, weil ich überall die gleiche Operation mache, kann ich also davon ausgehen, dass die Ausführungszeit dieselbe ist. Wenn nicht, muss ich auf den langsamsten Wetter warten. Das heißt, ich muss dann Pipeline

Stalls, und die sind natürlich teuer, die muss ich dann ausführen. Und auch so ein Out-of-Order ist nicht möglich, da ich vorher statisch schedule, werden Folge wie im Programm ausgeführt und auch durch die Pipeline durchgeschickt und dementsprechend können sich da einzelne Befehle innerhalb der Pipeline auch nicht überholen. Ich brauche dann auch keine Retirement Unit, der diese Ordnung wiederherstellt. Also insgesamt ist die Hardware natürlich einfacher. So ein Hardware Scheduling ist immer komplex und hat Potenzial. Allerdings, und das hat sich dann auch in der Realisierung gezeigt, ist gleiche operationen habe ja also zum beispiel wenn ich irgendwelche pixel operationen in großen bildern oder sowas habe und das hat dann auch dazu geführt dass tatsächlich wenn man sich die implementierungen anschauen die erfolgreichen implementierungen das waren dann doch eher so spezialprozessoren ja also für signal processing für multimedia processing die da bekannt sind und dann hat man natürlich irgendwann mal versucht intel und unit packard so ein joint venture versucht, okay, wir versuchen jetzt mal beide Vorteile zu nutzen, also die Vorteile einer Superskalaren Architektur in Kombination mit einer VLEW Architektur, um möglichst leistungsstarke Serverprozessoren zu bauen. Diese Variante hat dann EPIC geheißen, also Explicitly Parallel Instruction Computing, wurde realisiert in der sogenannten IA64 Architektur, auch besser die versucht haben beide Merkmale zu vereinigen, also sowohl ein Hard-Based Scheduling als auch Very Large Instructional-Architektur. Ist zumindest kommerziell gnadenlos schief gegangen, weil es war mal wieder der Versuch der eierlegenden Wollmilchsau und aus anderen Bereichen wissen wir ja, dass das immer ein bisschen schwierig ist mit dieser eierlegenden Wollmilchsau, dass die dann alles ein bisschen kann, aber nichts so richtig gut kann. So, also EPIC steht für Explicitly Das heißt, man hat versucht in der IA64 Architektur drei 64-Bit-Maschinenbefehle in einem Befehl zu zusammenzufassen. Und man hat also teilweise versucht, dieses dynamische Hardware Scheduling zum statischen Compiler Scheduling umzubauen. Also man hat versucht, die Hardware ein bisschen zu vereinfachen, einen einfacheren und schnelleren Hardware Scheduler, weil ein paar Aufgaben, die ein ansonsten Hardware Scheduler hat, hat man statisch, also in die Software, in den Compiler verlagert. dynamisches Scheduling bereitgestellt. Da sehen wir schon das erste Problem. Also ich stelle da jetzt eine Hardware zur Verfügung, die ein bisschen Hardware Scheduling kann, aber ein bisschen Verantwortung packe ich dann in den Compiler. Das heißt, ich verschiebe einfach die Probleme ein bisschen. Also einen Compiler für so eine Architektur zu erstellen, das ist kein Spaß. Also da ist wirklich Know-how gefragt, um einen Compiler zu bauen, der wirklich die vorteilhaften Eigenschaften dieser Architektur ausnutzt. raus haue, das ist kein Problem, aber ich brauche einen smarten Compiler, der eben diese VLEW-Architektur besonders gut nutzt. Und dann sage ich, naja gut, du Compiler, du kannst das eh nicht so richtig gut, also mache ich trotzdem ein bisschen dynamisches Scheduling. Also die Idee, die Grundidee war, dass der Compiler eine Voranalyse macht und versucht zu ermitteln, ok, wie viel Parallelität steckt da drin, wie viel kann ich davon statisch nutzen, das Ganze mache ich dann statisch, packe das also, diese drei Instruktionen in Und den Rest überlasse ich der Hardware. Jetzt habe ich hier Parallelität auf zwei Ebenen. Einmal statisch,

einmal dynamisch. Da muss ich erstmal entscheiden, okay, was packe ich wohin. Und insgesamt ist natürlich die Frage, steckt denn in meinen Anweisungen so viel Parallelität drin, dass ich das auch wirklich nutzen kann. Und das ist eben was, was nicht unbedingt so ganz der Fall ist. das ist was ganz Neues, da sprechen wir auch noch drüber, man versieht praktisch Instruktionen mit Prädikaten, also zum Beispiel besonders gut parallelisierbar oder keine Datenabhängigkeiten, also man packt da irgendwelche Prädikate dazu, die einen Hinweis darauf geben, ob das jetzt gut zu parallelisieren ist oder nicht. Man braucht sehr viele Register, um das auszuführen, man macht spekulative Ladebefehle, raus ja ich denke mir okay den befehl werde ich demnächst brauchen also lade ich den mal dass er da ist oder die daten werde ich demnächst brauchen haben wir schon beim cache kennengelernt ist eigentlich ein übliches verfahren und dann machen wir das spekulativ wenn ich natürlich zu viel spekulieren oder fehlt spekuliere dann kann das auch strafe kosten dann kann ich auch wieder langsamer werden und die idee war dass man so ein prozessor nicht für desktop baut sondern für high wenn ich für einen Rechner zu Hause für einen Prozessor über 2000 Euro verlange, dann wird sich den niemand kaufen, vermutlich. Gut, es gibt da so einen Journalisten, der das ganz gut zusammengefasst hat, was dabei rausgekommen ist, als er mal so einen Bericht über Nitanium verfasst hat, er hat dann einfach trocken geschrieben, Okay, this continues to be one of the greatest fiascos of the last 50 years. die Geschichte. Okay, ich glaube, das war ein gutes Schlusswort zu dem Thema Superskalarität.