Unser nächstes Thema, was wir in der Vorlesung behandeln wollen, das sind jetzt zwei Kapitel, die behandelt Superskalalität. Worum geht es hier? Pipelining, also Datenpfad ganz gut verstanden haben und eben auch Datenpfad mit der Simplen Pipeline verstanden haben und es geht jetzt erstmal darum, dass sie verstehen, warum man überhaupt Superskalarität eingeführt hat, warum Superskalarität zu Vorteilen führt und vielleicht auch, wo die Grenzen der Superskalarität sind und Superskalarität führt natürlich also Very Large Instruction World Machine, beziehungsweise EPIC, also das ist ja eine Unterart von der Very Large Instruction World Machine ist. Und diese zwei Architekturen wollen wir uns nochmal anschauen und vielleicht verstehen wir dann am Ende des Kapitels, warum in Spezialfällen diese Architekturen ganz erfolgreich sein können, aber warum sie im General Purpose Einsatz vielleicht doch nicht die Architekturen der Wahl sind. die wir einfach uns genauer anschauen müssen. Was wollen wir machen in diesem Kapitel? Wir wollen erstmal schauen, wir haben bis jetzt nur ganz einfache Pipelines besprochen und wir wollen diese Beschränkungen der einfachen Pipelines ein bisschen aufweichen, damit wir überhaupt Superscalar arbeiten können, Pipelining und Superscalarität verbinden können miteinander. Wir wollen uns ein paar Fallbeispiele anschauen, wie der PowerPC das macht. Wir wollen dann eine genauere Definition von Superscalarität erarbeiten und dann eben Very Large Instruction Word und EPIC dann nochmal miteinander vergleichen. Okay, worum geht es? Superskalalität zielt darauf ab, dass ich nicht einen Befehl lade, sondern dass ich gleichzeitig mehrere Befehle aus einem Befehlsstrom lade und bearbeite. Funktionale Einheiten, die voneinander unabhängig sind und die gleichzeitig Befehle verarbeiten können. Diese funktionalen Einheiten muss ich natürlich korrekt speisen mit Befehlen. Ich muss auf Datenabhängigkeiten achten, dass da nichts passiert, wenn ich parallel arbeite, dass am Ergebnis hinten das Ende rauskommt. Aber dieses Ergebnis möchte ich eben, wenn es geht, schneller haben, dadurch, dass ich parallel arbeite. Und wir haben bis jetzt eigentlich unsere Pipelines möglichst einfach gehalten. so ein paar Einschränkungen gehabt und diese Einschränkungen heißen zum Beispiel, dass ich die Execute-Phase immer in einem Taktzyklus abarbeiten kann. Das ist natürlich in der Realität nicht realistisch, weil so eine einfache Shift-Operation oder so eine einfache Addition, die dauert natürlich noch einen Taktzyklus oder vielleicht auch zwei Taktzyklen, aber eine komplexe Operation, also zum Beispiel eine Floating-Point-Division wie Wurzel ziehen oder sowas, das dauert natürlich deutlich länger und ich muss also jetzt für meine Pipelines auch die Möglichkeit vorsehen, dass eben die Execute-Phase zum Beispiel mehr als eine Taktzyklus dauert. Wir haben auch bis jetzt immer in unserer Pipeline nur eine Execute-Unit gehabt, also eine ALO, die eben unsere ganzen Operationen ausgeführt hat. habe das heißt diese einschränkung wird auch aufgehoben so dann haben wir noch die lineare eingabe das heißt das geordnet schön ein befehl nach dem anderen kommt auch darüber müssen wir reden weil wenn befehle datenabhängigkeiten beinhalten dann kann ich die nicht parallel bearbeiten also ich muss möglicherweise meinen code da ein bisschen umsortieren und out of order noch machen können muss, damit ich über Superskalarität überhaupt sprechen kann, ist, dass zwangsläufig mehr Konflikte auftreten können, die ich

lösen muss. Das heißt, dass ich nur einen Befehlskonflikt pro Taktzyklus betrachte, diese Forderung können wir auch nicht aufrechterhalten, die müssen wir auch noch auflösen. Wir werden es aber schön langsam nach und nach machen, eine Beschränkung nach der anderen auflösen. dann drehen wir uns mal auf die Einschränkungen 1 und 2, das heißt, dass wir nicht mehr genau einen Taktzyklus für die Execution Unit haben und dass es mehr als eine Execution Unit gibt. Das ist leicht zu realisieren. Wir bauen jetzt spezielle Execution Units, die eben ihre Aufgabe besonders gut lösen können. Also wir bauen eine spezielle Execution Unit, die eben eine Multiplikation schnell machen kann, schneller ausführen können. Es gilt natürlich immer das Prinzip, dass man nicht unnötige Hardware großartig aufbaut. Das heißt, im Idealfall schaffe ich es, die Execution Unit so gut wie möglich auszulasten. Das heißt, ich muss mir natürlich genau Gedanken machen, wie viele Multiplikationseinheiten brauche ich, wie viele Additionseinheiten brauche ich. Ja, wieder durch Tests einfach feststellen für meine Architektur, wie viele dieser Einheiten execution units sehr unterschiedliche aufgaben haben kann eben die anzahl an taktzyklen die die brauchen um die aufgabe zu lösen sehr unterschiedlich sein das heißt in der regel werde ich es schaffen irgendwelche integer aufgaben in einem taktzyklus zu lösen während eben komplexere operationen wie die floating point division bis zu 50 taktzyklen dauern kann das ist natürlich ein problem wenn ich mit pipelining arbeite möchte ich natürlich mit dieser operation division jetzt Ok, also hier einfach mal so, damit man es ein bisschen einordnen kann, so ein paar Beispiele. Wenn ich jetzt von einer Integer-Einheit spreche, dann macht die Integer-Einheit Loads und Stores. Also Loads und Stores über Integer-Register und Floating-Point-Register muss verfügt werden, die dann entsprechend mit Daten aus dem Hauptspeicher versorgt werden. Dann habe ich meine ganz normalen Integer-Allo-Operationen, also Addition, Subtraktion, logisches oder, logisches und, außer Multiplikation und Division, die lagere ich aus in die spezialisierte Einheit. Und dann muss ich natürlich Sprünge verarbeiten können. Für die Adressrechnung brauche ich dann auch eine spezialisierte Execution. Gut, die Multiplikationseinheit, die soll dann sowohl für Floating Point als auch für ganze Zahlen ungefähr 10 Zaktzyklen dauern. zwischen 20 und 50 Taktzyklen und so ein Floating Point Addierer, ein schneller Addierer für Floating Point, der soll das in 5 Taktzyklen starten. Also Sie sehen schon, da sind durchaus unterschiedliche Ausführungszeiten für unsere Execution Units vorgesehen. So und wie ist die prinzipielle Idee? Die prinzipielle Idee ist, ich habe nach wie vor meine 5 Phasen, also Instruction, Fetch, Decode, Execute, Memory und Writeback Phase. läuft eben nicht ein Befehl nach dem anderen, diese fünf Phasen, sondern mehrere Befehle gleichzeitig. Und in der ID-Phase werden diese Befehle aufgeteilt auf die vorhandenen Execution Units, also auf die Integer Unit, Multiplikations Unit, Floating Point Addition und Divisions Unit. Und die arithmetische Einheit, davon gibt es auch mehrere, oder die kann mehrmals pro Befehl genutzt werden. als ein befehl befindet sich auf so eine execution unit und das führt also dazu dass eine execution unit immer dann wieder frei ist wenn der vorherige befehl abgeschlossen ist ja also ein neuer befehl kann hier nur zugeordnet werden das heißt issue im englischen also ein issue kann nur ausgeführt werden wenn der befehl vorher erledigt ist und ich habe

natürlich so ein backlog in id also dass hier zwischen instruction fetch und id ein überholen noch nicht möglich ist also ich achte darauf dass die reihenfolge hier eingehalten wird das heißt hier kommen die befehle nacheinander an die werden aber natürlich parallel verarbeitet aber hier wird kein überholen passieren und hier werden sie dann parallel verarbeitet und hier hinten müssen die natürlich in order dann die schweigen. Das Problem, das wir jetzt haben, ist, dass für unsere Pipeline natürlich das Konfliktfenster deutlich größer wird, weil eben unsere Divisionseinheit bis zu 50 Taktzyklen dauern kann und dementsprechend muss ich das Konfliktfenster natürlich so groß machen, dass ich alle Befehle, die innerhalb dieser 50 Taktzyklen ausgeführt werden, eben auf Konflikte untersuchen muss. Ich brauche also nicht unsere einfache Konflikterkennung, wie vorher gebaut haben, sondern die wird deutlich aufwändiger und es gibt eben noch zusätzliche Ressourcenkonflikte, die in Write-Back-Phase möglich sind. Also das sind Konflikte, die wir bis jetzt noch nicht betrachtet haben, zum Beispiel der Write-After-Write-Konflikt. Das konnte bis jetzt nicht passieren, weil wir angenommen haben, dass die Execute-Phase immer genau einen Taktzyklus dauert. Aber wenn wir uns jetzt dieses einfache Beispiel hier anschauen, wir und die zwei Floating Point Register sind meine zwei Operanten hier, F2 und F4 und danach im nächsten Befehl kommt eine Subtraktion, auch hier erlandet das Ergebnis in F0 und es kann natürlich sein, dass die Subtraktion jetzt hier schneller fertig ist, das Ergebnis ablegt in F0 und danach dann die Division fertig wird und dieses Ergebnis wieder überschreibt. Das ist ein Write-after-Write-Konflikt, konnte bis jetzt nicht auftreten, bis jetzt haben wir nur Write-after-Read-Konflikte betrachten müssen, die wir ab jetzt auch betrachten und lösen müssen. Hier nochmal eben dargestellt, dass ein Write-After-Read nicht passieren kann, also weil wir eben das hier vorne Instruction-Federal-Instruction-Satz-ID dafür sorgen, dass das nicht passieren kann, eben länger dauert, dass dieses Write-Back nach diesem Write-Back passiert, da schreibe ich aufs Registerfall mein Ergebnis raus und damit kann ein Write-After-Write-Konflikt passieren. Write-After-Read ist momentan noch nicht möglich. Gut, wir haben ja schon bei der einfachen Pipeline über Konfliktlösungen gesprochen und die dass ich Wartezyklen einführe, aber Wartezyklen haben wir ja auch schon gelernt in einfachen Pipelines, das kostet natürlich Performance ohne Ende und je länger die Operationen in der Execute-Phase sind, desto teurer ist natürlich einfach Wartezyklen einzuführen. Ich kann jetzt nicht einfach, wenn ich so eine Division habe, dann 50 Wartezyklen einführen, um Konflikte zu vermeiden. legt sich bestimmte Architekturen und vermisst die dann, also wenn man zum Beispiel sich so einen MIPS Prozessor anschaut und die Floating Point Unit anschaut und dann so eine Spice Last also das ist ein Programm einfach, vermisst auf diese Architektur und dann sieht man schon, dass das Ganze eben nicht ganz trivial ist dass eben 35% aller Zyklen Wartezyklen sind, das heißt man muss performance technisch in der Parallelität schon was rausholen schaden als sie nutzen. Gut, das Problem ist natürlich die lineare Eingabe, die wir derzeit betrachten und deshalb wollen wir auch diese Einschränkung aufheben, wir wollen also diesen Rückstau, den wir erzeugen durch die lineare Eingabe, das sind also Execution Units frei, die eiteln und wir hätten Befehle, die vernünftig was machen könnten, allerdings kämen die Das heißt,

in gewissen Grenzen erlauben wir eine Umordnung, aber wir müssen natürlich schauen, dass wir durch diese Umordnung keine neuen Konflikte entstehen lassen und erlauben also eine nicht lineare Eingabe. Das heißt, wir ordnen quasi Befehle um und diese Befehle, die sollen eben die entstehenden Wartezyklen minimieren. pipeline sieht dann folgendermaßen aus wir haben hier ein paralleles instruction fetch also mehrere instruktionen werden gleichzeitig geholt die instruktionen werden dekodiert und dann gleich noch ein bisschen die register entsprechend umbenannt damit hier schon mal die ersten groben konflikte gelöst werden dann werden die in so genannte instruction window vorgeladen also sind die nächsten instruktionen die zur ausführung anstehen wo man schon mal konflikt Dann haben wir hier so eine Issue Unit, also so eine Anweisungseinheit, die sorgt dafür, dass in der richtigen Reihenfolge dann diese Instruktionen verteilt werden auf die funktionalen Einheiten. Die funktionalen Einheiten selber verfügen nochmal über so einen kleinen Puffer, also über so eine Reservierungsstation, wo praktisch die auszuführenden Operationen hier anstehen, bis die Execution Unit frei ist. der eigenwillige Befehl ausgeführt wird, werden diese Ergebnisse geschrieben und hier muss ich meine Write-Back-Einheit natürlich deutlich erweitern, weil wir wollen die Ergebnisse in der richtigen Reihenfolge natürlich schreiben. Das heißt, hier wird nochmal auf Konflikte überprüft und auf Reihenfolge überprüft und die Ergebnisse dann in der richtigen Reihenfolge geschrieben. Das heißt also, im Vergleich zu einer einfachen Pipeline muss ich ein bisschen Umbenennung machen, also ein Rename, um eben so weit wie möglich Konflikte zu vermeiden, also Write-after-Write und Write-after-Read Konflikte zu vermeiden. Ich muss diese Issue-Einheit, die hat es ja bis jetzt nicht gegeben, die muss ich neu einführen. Das ist eben die Zuweisung von Befehlen zu den Execution Units. Und das nicht unbedingt unter Einhaltung der Reihenfolge, also Out-of-Order auch ermöglichen. also hier die ursprüngliche Reihenfolge wiederherstellen. Also Sie sehen, dass die Idee parallel zu arbeiten zwar erstmal gut klingt, dass die zu realisieren aber doch ein bisschen aufwendiger wird, dass also aus einer einfachen Pipeline dann doch schon eine etwas schwierigere Pipeline wird. Ja, das heißt also, wir haben in unserer Pipeline jetzt zwei Bereiche, zur issue unit da wird linear ein befehl nach dem anderen geholt dekodiert und auch so ein renaming durchgeführt wenn wir dann auf den execution unit sind dann kann es durchaus nicht linear sein also out of order reihenfolge sein und diese linearität die wird dann später in der retire and writeback phase wieder hergestellt also wir haben einen inorder bereich gefolgt von einem out of order hergestellt. Das Ganze natürlich in Hardware, davon bekommt der Benutzer nichts mit. Einzige, der vielleicht ein bisschen davon betroffen ist, ist der Compiler, den ich für meine Programme erstelle, aber ansonsten wird ja sehr viel in Hardware passieren und wie das genau funktioniert in der Hardware werden wir dann im nächsten Kapitel auch noch ein lebt das ist zum beispiel der ibm power pc so ein block schaltbild von diesem ibm power pc enthält schon mal alles sagen wir mal was rang und namen hat an technologie also wir haben hier ein branch target address cache untergebracht wir haben den branch history table wir haben return address tag also hier oben ist der instruction cache und wenn man die roten pfeile ist es wie so eine ganz normale pipeline das heißt ich hole mir meine instruktion ich

decodiere die ich so wie wir es auf der folie vorher gesehen haben ich mache so ein renaming damit ich konflikte vermindere ich schreibe das ganze in den puffer und auf diesem puffer arbeitet dann die instruction issue unit und die hat unter sich eben die parallel arbeitenden execution units und die das heißt wir haben hier einen Instruction Cache, wir haben hier einen Data Cache und über ein Bus Interface Unit ist es dann quasi mit dem Hauptspeicher und mit den anderen verbunden. Die roten Pfeile ist der Befehlsfluss, Instruction Fluss und die blauen Pfeile, das ist der Datenfluss. So und PowerPC, woher die Abkürzung kommt, da steht hier nochmal Performance Optimization with Enhanced Risk and PC für Performance Chip. Okay, mit diesem Bild möchte ich für heute enden.