Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ETH Institut für Verhaltenswissenschaften

Design eines programmierbaren Rechners

Ein Leitprogramm für Fachhochschüler der Gebiete Informatik und Elektrotechnik

Bearbeitungsdauer : ca. 30 Stunden

Verfasst von Markus Sauter
Betreuung durch Prof. J. Hromkovic
ETH Zürich, im September 2005

Praxisstatus: Nicht erprobt

Liebe Leserin, lieber Leser

Sie halten ein Leitprogramm mit dem Titel „ Design eines programmierbaren Rechners“ in Ihren Händen. Doch um welchen Inhalt geht es hier genau? Vielleicht ergeht es Ihnen im Studium des elektronischen Wunders „Computer“ ähnlich wie mir: Einerseits lernt man wie eine einfache logische Addiererschaltung aufgebaut ist, wie ein Multiplexer funktioniert oder was ein Speicherbaustein alles beinhaltet. Auf der anderen Seite werden viele Programmiersprachen erlernt, Datenstrukturen implementiert und Paradigmen umgesetzt. Doch was passiert dabei im Rechner genau? Wie kann ein Computer die Programmieranweisungen mit den vorhandenen Hardware-Komponenten umsetzen? Das Leitprogramm versucht diese Schnittstelle zwischen Hard- und Software zu erläutern, indem eine minimale, programmierbare Rechnerstruktur exemplarisch erarbeitet wird. Um die einzelnen Schritte einfach zu halten, wird kein Wert auf Optimalität der Rechnerstruktur gelegt. Das Ziel ist, beispielhaft die grundlegendsten Konzepte eines programmierbaren Rechners aufzuzeigen um als Einblick und Einstieg in die höheren Künste der Rechnerstrukturen zu dienen.

Voraussetzungen

Im Leitprogramm wird versucht mit minimalen Mitteln einen programmierbaren Rechner zu entwerfen. Alle dafür benötigten Hardware-Komponenten werden kurz eingeführt, dies kann jedoch nicht bis ins letzte Detail geschehen. Auch auf Software-Seite werden nur die nötigsten Assembler-Anweisungen (Programmierbefehle) vorgestellt, aber auch hier kann nicht alles von Grund auf erarbeitet werden. Um den grösstmöglichen Nutzen aus diesem Leitprogramm zu ziehen, sollte der interessierte Leser mit logischen Schaltungen Erfahrung haben, das Rechnen mit binären Zahlenwerten im Blute haben, sowie die einzelnen Hardware-Komponenten bereits etwas kennen. Auch sollte das Programmieren in Assembler kein komplettes Neuland sein, damit das Hauptaugenmerk auf die Abläufe in unserem Rechner gerichtet werden kann. Falls Sie diese Voraussetzungen nicht ganz erfüllen, trotzdem aber Interesse an der Thematik des Leitprogramms haben, dann dürfen Sie gerne einen Versuch starten. Das Leitprogramm sollte trotzdem verständlich sein und einen guten Einblick vermitteln können, auch wenn nicht alle Details zu den einzelnen Bereichen bekannt sind.

Inhaltliche Gliederung

Das Leitprogramm ist in 6 Kapitel unterteilt. **Das erste Kapitel** beschäftigt sich mit den für den Rechner benötigten Hardware-Komponenten: es wird eine sehr einfache ALU (Arithmetic Logic Unit) entwickelt, die Register und der Hauptspeicher werden vorgestellt aber auch weitere, kleinere Bausteine wie Multiplexer oder Shift-Extension werden wiederholt. Am Schluss des Kapitels werden bereits zwei einfache Schaltungen z.B. eine für den „Program-Counter“ eingeführt worden sein. **Das zweite Kapitel** hat die Software-Seite zum Inhalt. Es werden die wenigen von uns benötigten Assembler-Befehle vorgestellt und diese in sogenannte „Format-Klassen“ unterteilt. Wir betrachten dabei insbesondere diejenigen Assembler-Befehle, die dann direkt von der Rechnerstruktur verarbeitet werden können. Diese Befehle bilden das sogenannte „Instruction-Set“ unseres Rechners. Im Anschluss werden mit einigen Beispielen und Übungen die Programmverzweigungen und Schleifen etwas aufgefrischt, und danach wird das Konzept der Prozeduren und deren Management im Datenspeicher mittels eines sogenannten „Stack“ besprochen. **Im 3. Kapitel** geht’s dann richtig zur Sache: Es wird gezeigt, wie aus den Assembler-Anweisungen des Instruction-Sets ein „Maschinencode“ speziell für unsere Rechnerstruktur erzeugt werden kann. Der Maschinencode ist die Darstellung eines Befehls, die von der Rechnerstruktur „verstanden“ und verarbeitet werden kann. Der Maschinencode bildet somit die Schnittstelle zwischen Hard- und Software. **Im Kapitel 4** beginnen wir unseren Rechner zusammenzubauen. Es wird gezeigt, welche Leitungen die Daten bei einer Anweisung (Maschinencode-Instruktion) im Rechner durchwandern müssen. Die Rechnerstruktur wird dabei schrittweise erweitert, bis alle Anweisungen unseres Instruction-Sets ausgeführt werden können. **Im 5. Kapitel** wird dann dem Rechner das Leben eingehaucht: Es wird aus einzelnen Signal-Schaltungen die „Control Logic“-Komponente für den Rechner erarbeitet. Diese hat zur Aufgabe alle Rechnerkomponenten so zu steuern, dass die Daten aus einer Instruktion auch wirklich die vorgesehenen Datenpfade durchwandern. Mit dem Einbau der Control Logic wird unsere Rechnerstruktur vervollständigt worden sein. In **Kapitel 6** werden die gemachten Schritte zur Erarbeitung des Rechners nochmals zusammengefasst. Mit Erläuterungen zu einigen angenommenen Vereinfachungen soll unsere Rechnerstruktur auf Vor- und Nachteile ausgeleuchtet, und das erhaltene Bild zu der Rechnerstruktur dadurch abgerundet werden.

Zur Arbeit mit dem Leitprogramm

Sie finden in jedem Kapitel einige Übungsaufgaben, über die Sie den Inhalt des Kapitels vertiefen und ihr Verständnis kontrollieren können. Um dabei nicht immer durch das Leitprogramm blättern zu müssen, nutzen Sie den Anhang A, in dem Sie eine Zusammenstellung der wichtigsten Tabellen und Informationen finden. Für die Aufgaben benötigen Sie sonst nur einen Stift und Papier. Ein Taschenrechner mit Funktion für die Umrechnungen von Binärzahlen zu Dezimalzahlen kann aber einiges an Zeitaufwand einsparen helfen. Bei einigen Aufgaben werden noch Kopien einer Vorlage aus dem Anhang benötigt. Zur Selbstkontrolle finden Sie in Anhang B die Musterlösungen zu allen gestellten Übungsaufgaben. Haben Sie die Aufgaben alle gelöst, so sollten Sie keine Probleme haben, bei dem Kapiteltest am Ende jedes Kapitels zu zeigen, dass Sie den Inhalt des Kapitels verstanden haben.

Inhaltsverzeichnis

Kapitel 1: Hardware-Komponenten

Einleitung

Wir wollen nun als erstes die Hardware-Komponenten im Einzelnen betrachten. Dabei wird jeweils auch die Beschriftungskonvention dieses Leitprogramms zu den Hardware-Komponenten vorgestellt. Das Kapitel ist in 4 Unterabschnitte gegliedert:

1 Im ersten Abschnitt betrachten wir die Darstellung von Signalleitungen sowie erste einfache Komponenten wie Shift und Sign-Extension.

2 Danach wiederholen wir logische Gatter und bauen uns einen Multiplexer. Mit den gleichen wenigen Gattern können wir uns auch einen Addierer zusammensetzen, und werden diesen um Logische Operationen, Subtraktion und um Vergleichsoperationen erweitern. Diese Komponente wird uns bereits als ALU (Arithmetic Logic Unit) genügen.

3 Was dann noch fehlt sind Register und der Hauptspeicher, also Komponenten, die Zahlenwerte zwischenspeichern können. Diese Speicherbausteine benötigen ein „Clock"-Signal als Taktgeber, auf die Clock werden wir aber nur am Rande eingehen.

4 Zum Abschluss des Kapitels werden in zwei Beispielen einige der Komponenten zu einer Schaltung kombiniert. Wir setzen uns einen „Program-Counter" zusammen, der uns Maschinencode-Instruktionen sequentiell aus einem „Instruction Memory" ausliest. Als zweite Schaltung schliessen wir vereinfacht die ALU an einen „Register-Block" an.

1.1 Signalleitungen, Shift left 2 und Sign-Extension

Was wir als erstes benötigen ist die Möglichkeit, Daten durch Signale irgendwie zwischen den Komponenten auszutauschen. Wir nehmen dafür einen Drahtleiter und können bereits das Signal 0: kein Strom fliesst durch den Leiter, und das Signal 1: es fliesst Strom, übertragen. In Abb. 1.1 wird ein Leiter durch einen gerichteten Pfeil dargestellt. Es kann also ein binäres Signal, der Wert 0 oder 1, von Komponente A zu Komponente B übertragen werden.

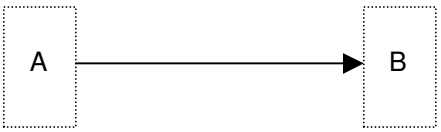


Abb. 1.1: Ein Signal wird über einen Leiter von Komponente A zu Komponente B übertragen. Ein einzelner Leiter kann die Signalwerte 0 oder 1 annehmen.

Falls sich eine Leitung/ein Signal in mehrere Leitungen verzweigt, wird dies durch einen Punkt auf der Verzweigung markiert.

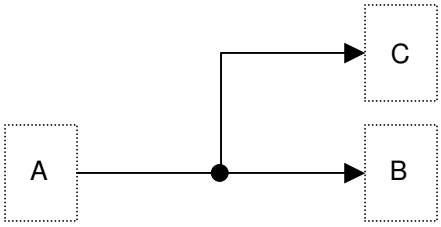


Abb. 1.2: Eine Leitung von Komponente A aus kommend wird verzweigt und in die Komponenten B und C eingegeben.

Um komplexere Signale zwischen den Komponenten austauschen zu können, werden mehrere Leiter parallel verwendet. In Abb. 1.3a) werden 5 parallele und mit einem Index versehene Datenleiter gezeigt. Die Indexierung beginnt dabei immer bei 0. Diese können dadurch 2 hoch 5 = 32 unterschiedliche Signale übertragen. Man sagt: „der Leiter oder das Signal ist 5 Bit breit“. Normalerweise werden wir 32 Bit breite Leiter verwenden. Dadurch sind 2 hoch 32 = 4’294’967’296 unterschiedliche Bitmuster (binäre Signale) möglich.

a) b)

Abb. 1.3: In a) sind 5 einzelne, indexierte Leiter für die Übertragung eines 5 Bit breiten Signals abgebildet. In b) ist die verkürzte Schreibweise dargestellt.

Natürlich können sich auch parallele Leitungsstränge verzweigen.

a) b)

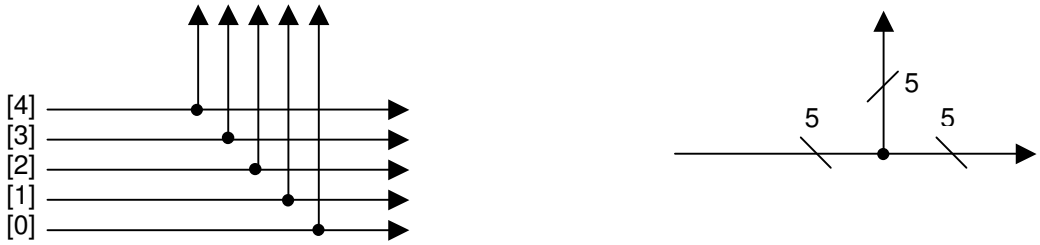


Abb. 1.4: In a) ist die Verzweigung aller einzelnen indexierten Leitungen zu sehen. In b) ist die vereinfachte Schreibweise dargestellt.

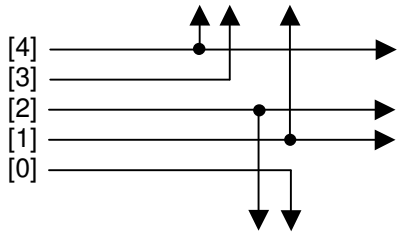
Werden bei einer Verzweigung mehrerer Leiter nicht alle Leitungen abgezweigt, dann sind die jeweiligen Leitungen durch den Index angegeben.

a) b)

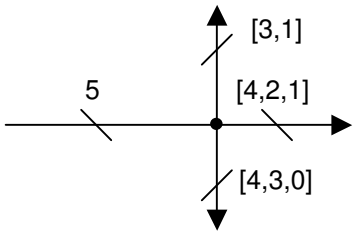
Abb. 1.5: In a) ist die Verzweigung von einzelnen indexierten Leitungen abgebildet. In b) ist die abgekürzte Schreibweise dargestellt. Hier werden nach einer Verzweigung die Indizes der abgezweigten Leiter angegeben.

Aufgabe 1.1: Hier die erste Übungsaufgabe: Zeichnen Sie die Leitungen bei a) in der abgekürzten Schreibweise, und bei b) die Leitungen in der ausführlichen Schreibweise auf.

a)

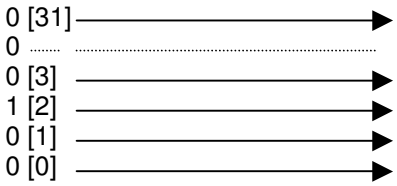


b)



Wollen wir auf einem Leiter einen konstanten Wert (0 oder 1) einspeisen, so wird der Wert 0 oder 1 vor dem Leiteranfang angegeben. Ein konstanter Wert auf mehreren parallelen Leitern wird durch einen Dezimalwert am Leiteranfang angegeben. Dieser Wert kann nun einfach in einen binären Wert umgewandelt und die Bitstellen in die Leiter mit entsprechendem Index eingespiesen werden. In Abb. 1.6 wird in 32 parallele Leiter der Wert 4 eingegeben: 00000000000000000000000000000100. Das „Least Significant Bit“ ist der Leiter mit dem Index 0, das „Most Significant Bit“ (das Meistwertige Bit) findet man in der Leitung mit dem höchsten Index.

a)



b)

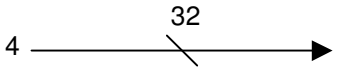
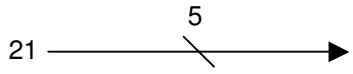


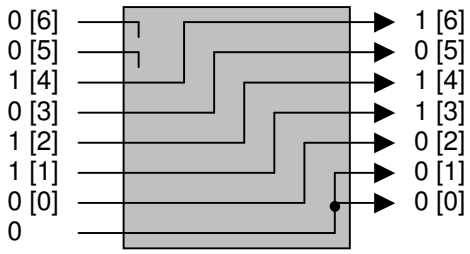
Abb. 1.6: In a) ist der Dezimalwert 4 in einen Binärwert umgewandelt worden. Die einzelnen Leiter werden mit der Konstante 0 oder 1 der jeweiligen Bitstelle gefüllt. Das Least Significant Bit befindet sich im Leiter mit Index 0. In b) ist die verkürzte Schreibweise mit einem Dezimalwert vor einem Leiteranfang abgebildet.

Aufgabe 1.2: Zeichnen Sie nun in der ausführlichen Darstellung eine 5-Bit breite Leitung, in die der konstante Wert 21 eingespiesen wird. Die zugehörige verkürzte Schreibweise ist unten gegeben.



Für den Rechner können wir uns bereits zwei einfache Komponenten bauen. Die erste dieser Komponenten ist ein sogenannter **Shift left 2** (Verschiebung nach links um 2 Stellen). Das heisst die beiden signifikantesten Stellen eines Binärwertes fallen weg. Alle anderen Bits werden um 2 Stellen nach links verschoben und die beiden neuen, am wenigsten signifikanten Stellen werden mit dem Wert 0 gespiesen. Wie beim Beispiel in Abb. 1.7 zu sehen ist, werden bei dem angelegten Wert 0010110 die ersten 2 Stellen [00] abgeschnitten. Es verbleibt 10110. Nun werden zwei 0-en hinten angehängt: 10110[00]. Das Resultat ist dann 1011000. Die Shift left 2-Operation entspricht einer Multiplikation des Binärwertes mit 4.

a)



b)

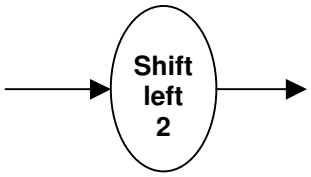
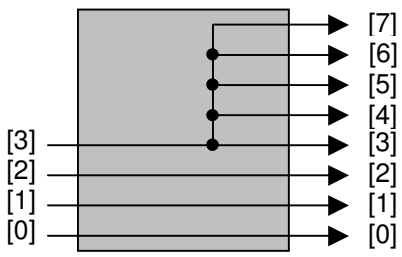


Abb. 1.7: In a) ist die Schaltung einer Shift left 2-Operation eingezeichnet. Als Beispiel wird ein 7-bit breites Signal mit dem binären Wert 22 um 2 Stellen nach links verschoben, was einer Multiplikation mit 4 gleichkommt. Das Resultat am Ausgang beträgt 88. In b) ist das Schaltsymbol für die Shift left 2 Komponente angegeben.

Aufgabe 1.3: a) Welche Bedingung wurde stillschweigend angenommen, damit das Shift left 2 wirklich einer Multiplikation mit 4 entspricht ? b) Welche Multiplikation würde einem Shift left 4 entsprechen?

Die zweite Komponente ist die sogenannte **Sign-Extension**. Falls z.B. ein 16 Bit breites Signal in 16 Leitungen übertragen wird, eine Komponente aber 32 Signalleitungen am Eingang erwartet, so müssen die 16 Leitungen auf 32 Leitungen erweitert werden. Die 16 neu hinzugekommenen Leiter werden mit dem Wert des bisherigen Most Significant Bit aufgefüllt (im Beispiel also der Leiter mit Index [15]). Der Grund dafür ist, dass das Most Significant Bit in der Zweierkomplement Darstellung von Binärwerten das Vorzeichen, also ob positiver oder negativer Wert, darstellt. Dieses Vorzeichen muss bei einer Sign-Extension beibehalten werden. Das Zweierkomplement eines binären Wertes wird beim Bau der ALU in diesem Kapitel nochmals angesprochen. In Abb. 1.8 wird mit einer Sign-Extension von 4 auf 8 Leitungen erweitert.

a)



b)

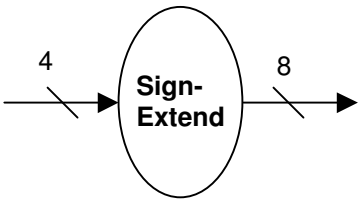


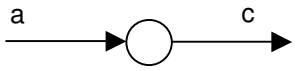
Abb. 1.8: In a) ist die Schaltung für eine Sign-Extension Operation gegeben. Es wird von 4 auf 8 Datenleitungen erweitert. In b) ist das Schaltungssymbol für die Sign-Extension Komponente abgebildet. Die Breite der ein- und ausgehenden Signale ist jeweils angegeben.

Aufgabe 1.4: Würde nur mit positiven binären Zahlen gerechnet werden, müssten die neu hinzukommenden Leitungen bei einer Sign-Extension das Most Significant Bit des binären Wertes nicht berücksichtigen. Nennen wir diese neue Komponente „Extension“. Zeichnen Sie solch eine Extension-Komponente für von 4 auf 8 Datenleitungen.

1.2 Multiplexer, Addierer und ALU

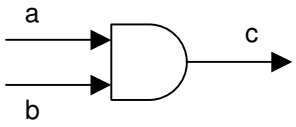
Als nächstes repetieren wir kurz die logischen Gatter AND, OR, und NOT. Wir gehen dabei nicht auf deren physikalischen Aufbau ein, sondern wir interessieren uns nur für die Wirkung am Ausgang c bei gegebenen Inputs a und b. Die Gatter sind in Abb. 1.9 zusammengefasst.

NOT



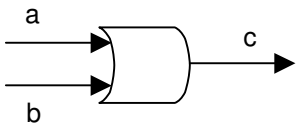
a	c
0	1
1	0

AND



a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

OR

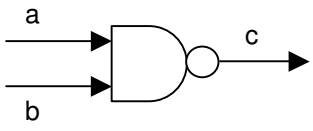


a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

Abb. 1.9: Die Schaltungssymbole für logischen Gatter NOT, AND und OR sind in der linken Hälfte abgebildet, rechts davon ist die dazugehörige Wertetabelle gegeben.

Im folgenden wird in einer Schaltung häufig das NOT-Gatter direkt an ein AND- oder OR Gatter angehängt. Dabei wird ein Pfeil beim NOT-Gatter zur Vereinfachung des Schaltbildes weggelassen. In Abb. 1.10 ist ein Beispiel dazu gezeigt. Die Schaltung ist übrigens als NAND-Gatter bekannt.

NAND



a	b	c
0	0	1
0	1	1
1	0	1
1	1	0

Abb. 1.10: Das Schaltungssymbol für das logischen Gatter NAND, bestehend aus einem AND-Gatter, an das direkt ein NOT-Gatter angehängt wurde. Bei diesem NOT-Gatter ist ein Pfeil in der Darstellung zur Vereinfachung des Schaltbildes weggelassen worden.

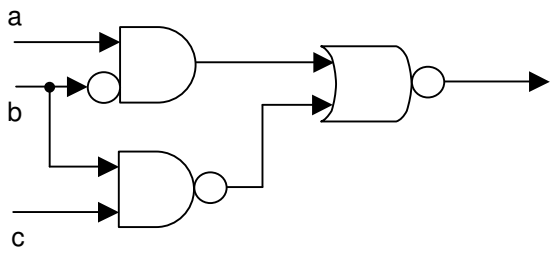
Aufgabe 1.5: Wir wollen nun das Erstellen von Schaltungen und ihren zugehörigen Wertetabellen etwas üben.

a) Zeichnen Sie eine Schaltung, die nur zu einer 1 am Output c auswertet, wenn die beiden Inputsignale a und b beide 0 sind. Geben Sie die Wertetabelle dazu an.

b) Zeichnen Sie eine Schaltung, die folgende Wertetabelle erfüllt. Die entsprechenden Schaltung wird XOR-Gatter genannt.

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

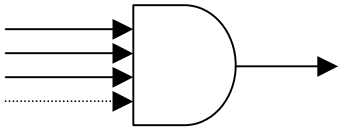
c) Geben Sie die Wertetabelle an, die durch folgende Schaltung erzeugt wird.



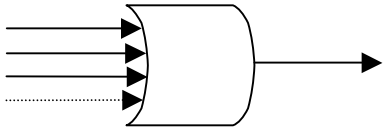
Aufgabe 1.6: Erstellen Sie eine Schaltung für ein Mehrweg-AND Gatter sowie eine Schaltung für ein Mehrweg-OR Gatter aus den vorgestellten AND und OR Gattern. Dies könnte zum Beispiel ein 3-Weg AND/ 3-Weg OR Gatter oder ein 8-Weg AND/ 8-Weg OR Gatter sein.

Das Mehrweg-AND und das Mehrweg-OR Gatter haben mehrere Eingangssignale. Nur wenn alle diese Eingangssignale gleich 1 sind, wird am Ausgang eines Mehrweg-AND Gatters eine 1 ausgegeben. Beim Mehrweg-OR Gatter ist die Ausgangsleitung gleich 1, falls mindestens eine der Eingangssignale gleich 1 ist. Wir werden für das Mehrweg-AND und das Mehrweg-OR Gatter folgende Schaltungssymbole benutzen:

Mehrweg-AND

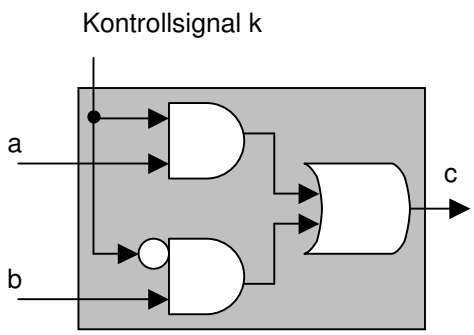


Mehrweg-OR



Mit den logischen Gattern können wir uns einen sogenannten **Multiplexer** zusammenbauen. Ein Multiplexer hat 2 Eingangsignale. Mit einem dritten Signal, dem Kontrollsignal, kann gesteuert werden, welches der Inputsignale durchgeleitet werden soll. Dies kann auch in der Wertetabelle zum Multiplexer, Abb 1.11b) gesehen werden: Ist das Kontrollsignal k auf 0, so ist der Ausgang c gleich dem Wert der Inputleitung b. Ist hingegen das Kontrollsignal k auf 1 gesetzt, so entspricht der Ausgang c dem Inputsignal a. Die Kontrollsignale, Signale die Komponenten steuern, werden im folgenden Grün dargestellt, Leitungen die Daten zwischen Komponenten übertragen, werden Blau eingefärbt.

a)



b)

a	b	k	c
0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1

c)

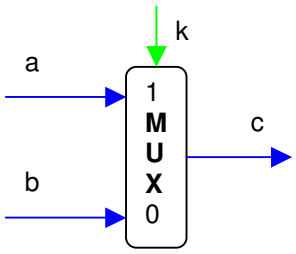
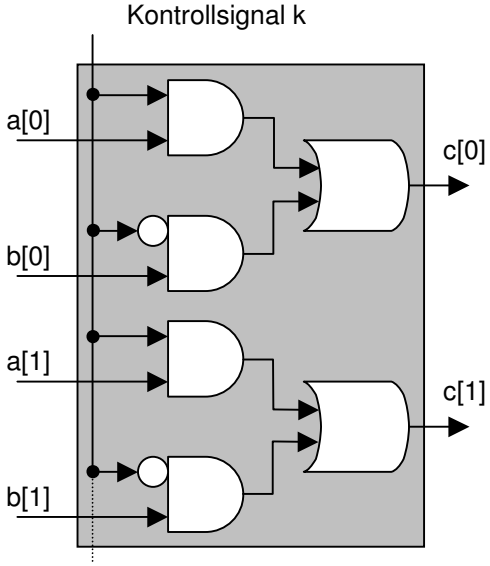


Abb. 1.11: In a) ist die Schaltung eines 2-Weg Multiplexer gegeben (2 Signale stehen zur Auswahl). In b), der Wertetabelle zu einem Multiplexer sieht man, dass wenn das Kontrollsignal k auf 0 steht, das Signal b an den Ausgang c angelegt wird. Ist das Kontrollsignal jedoch 1, so wird das Signal a an den Ausgang c geleitet. In c) ist das Schaltsymbol für den Multiplexer dargestellt.

Mit einem Multiplexer kann auch zwischen zwei komplexeren Inputsignalen eines ausgewählt werden.

a)



b)

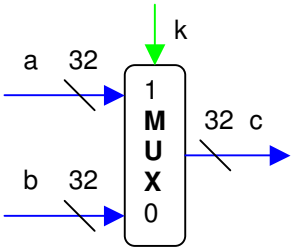
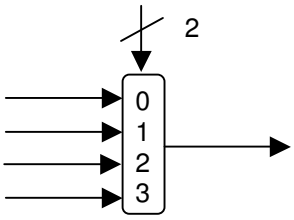


Abb. 1.12: In a) ist die Schaltung für einen Multiplexer mit 2 Bit breiten Inputsignalen a und b angegeben. Der Multiplexer kann durch Parallelschaltung von einzelnen Multiplexer gebildet werden. In b) ist das Schaltsymbol für einen 2-Weg Multiplexer mit 32-Bit breiten Inputsignalen abgebildet.

Aufgabe 1.7: Erstellen Sie aus dem vorgestellten Multiplexer einen 4-Weg Multiplexer. Dieser kann mit zwei Kontrollsignalen ein Signal aus 4 Signaleingängen auswählen und weiterleiten. Das verwendete Schaltungssymbol ist das folgende:



Unser nächstes Bauteil wird bereits der **Addierer** sein. Wenn eine Stelle eines Binärwertes addiert werden soll, so ist neben den Werten der Stelle auch ein eventueller Übertrag aus der vorhergegangenen Stellenaddition, das „Carry In", einzubeziehen. Weiter ist neben dem Resultat der Stellenaddition auch ein eventueller Übertrag, das „Carry Out", an die nächste Stellenaddition weiterzuleiten.

a	b	Carry In	Carry Out	Result
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

Abb. 1.13: Wertetabelle zu einem 1-Bit Addierer mit den Stellenwerten a und b sowie dem Übertragseingang „Carry In". Der Signalausgang „Carry Out" und das Stellenresultat werden durch die Signaleingänge eindeutig bestimmt.

Die Schaltung für einen solchen Einstellenaddierer könnte wie in Abb. 1.14 aussehen.

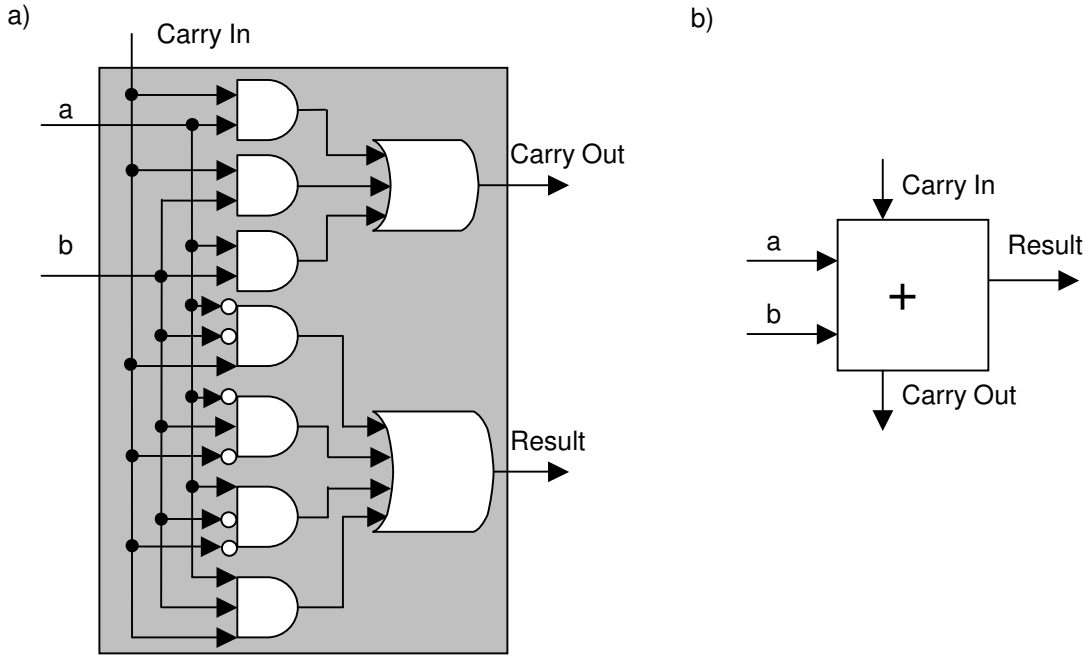


Abb. 1.14: In a) ist die Schaltung für einen 1-Bit Addierer abgebildet. In b) ist das Schaltsymbol für den 1-Bit Addierer dargestellt.

Um nun einen 32-Bit Addierer zu erhalten, können mehrere dieser 1-Bit Addierer parallel geschaltet werden. Beachten Sie, dass die Schaltung des 32-Bit Addierers aber rund 32 mal länger braucht als ein einfacher 1-Bit Addierer. Dies, weil der 1-Bit Addierer für z.B. die zweite Stelle warten muss, bis das richtige „Carry In“ für seine zweite Stelle vom ersten 1-Bit Addierer anliegt. Das „Carry In“-Signal läuft also seriell von der ersten bis zur letzten Stellenaddition, dadurch können nicht alle Stellenaddierer gleichzeitig schalten und deshalb wird die Schaltung so viel länger brauchen, bis überall die richtigen Resultate anliegen.

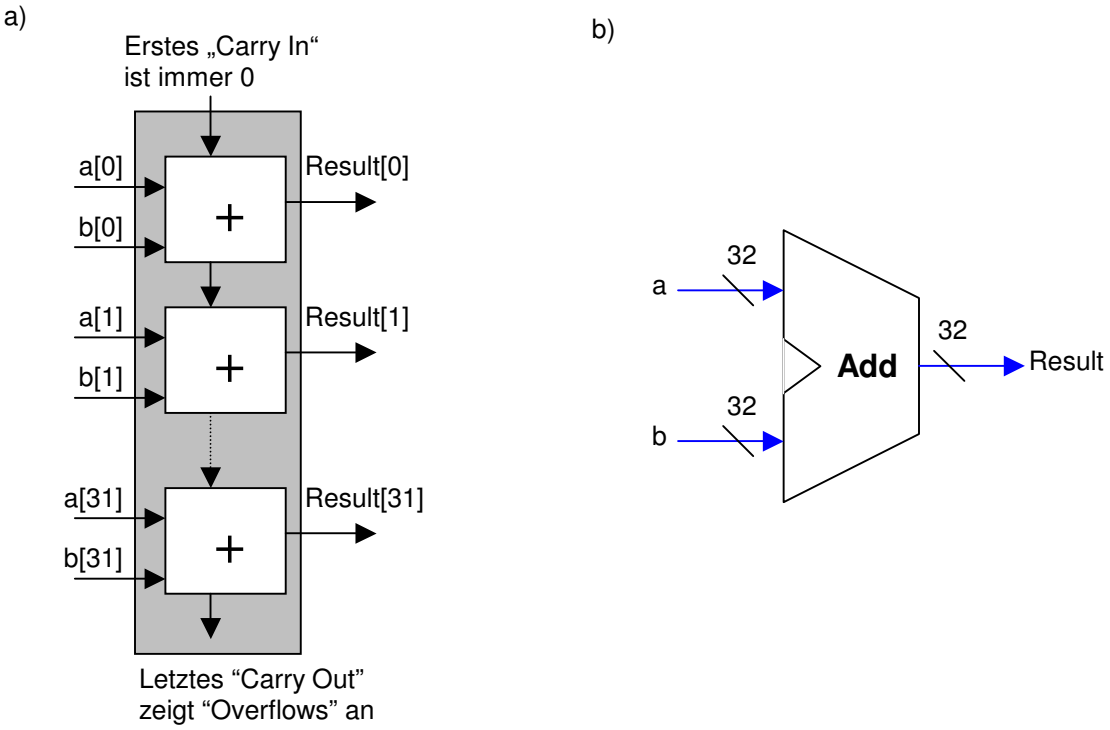


Abb. 1.15: In a) wird durch Parallelschaltung von 1-Bit Addierern ein 32-Bit Addierer zusammengesetzt. In b) ist das Schaltungssymbol für den 32-Bit Addierer zu sehen. Das letzte „Carry Out“ zeigt mit einer 1 an, dass die Breite des Resultats über die Breite des Addierers hinausgeht. Dies wird „ein Overflow“ genannt, der sofern er nicht weiter berücksichtigt wird, zu falschen Resultaten führt.

Aufgabe 1.8: Zeichnen Sie die vollständige Schaltung eines 4-Bit Addierers und beschriften Sie für die Addition von 4 + 5 sämtliche Leitungen mit den entsprechenden Werten.

Wir beginnen nun den Addierer schrittweise zu einer **ALU (Arithmetic Logic Unit)** zu erweitern. Der deutsche Begriff für die ALU ist „das Rechenwerk“. Wir bauen dafür zuerst eine 1-Bit ALU Komponente, in der durch einen 3-Weg Multiplexer mit zwei Kontrollsignalleitungen zwischen den Operationen Addition, und stellenweisem AND und OR ausgewählt werden kann. Die Schaltung ist in Abb. 1.16 auf der nächsten Seite abgebildet.

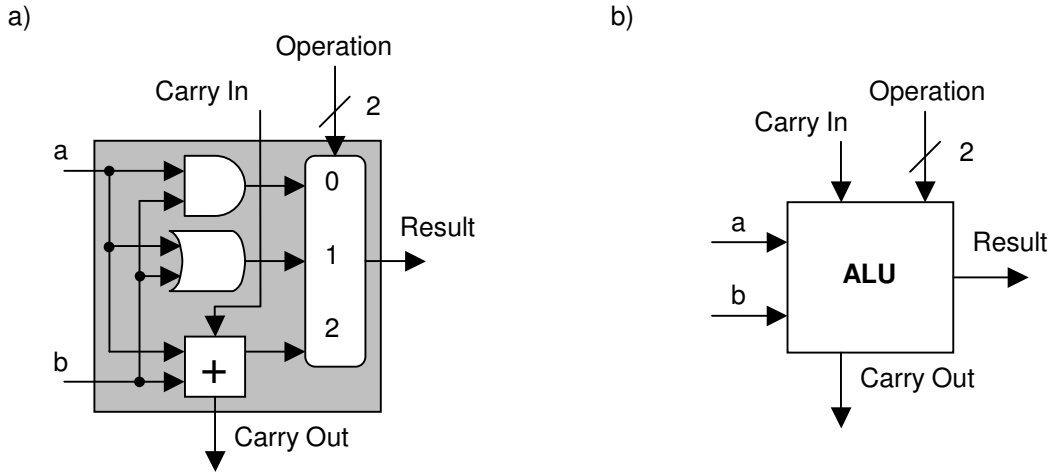


Abb. 1.16: In a) ist die Erweiterung vom 1-Bit Addierer zu einer 1-Bit ALU gezeigt. Durch den 3-Weg Multiplexer und dessen 2 Bit breites Kontrollsignal „Operation“ kann bestimmt werden, welches Ergebnis aus den Operationen AND, OR und Addition and den Ausgang „Result“ weitergeleitet wird. In b) ist das Schaltsymbol für die 1-Bit ALU gezeigt.

Die Parallelschaltung zu einer 32-Bit ALU für die Operationen AND, OR und Addition ist in Abb. 1.17 zu sehen.

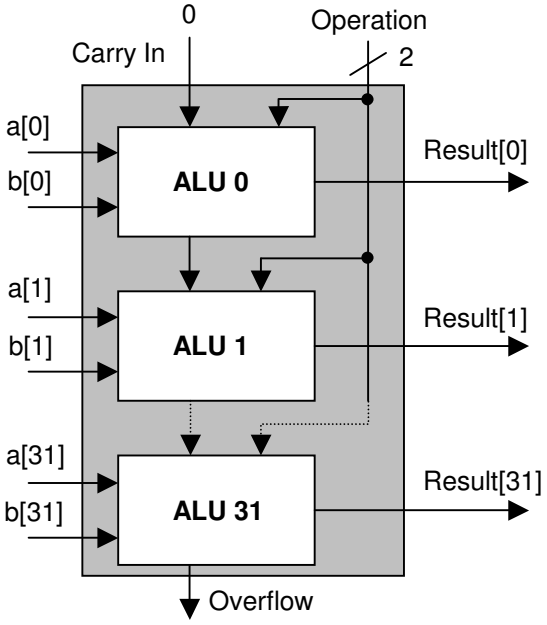


Abb. 1.17: Durch Parallelschaltung der 1-bit ALU Komponente erhalten wir eine 32-Bit ALU für die Operationen AND, OR und Addition.

Als nächstes wollen wir die ALU um die Subtraktionsoperation erweitern. Dafür eine kurze Wiederholung des Zweierkomplementes.

Das Zweierkomplement ist eine binäre Darstellungsart von positiven und negativen Integer-Zahlen (Ganze Zahlen) mit der sehr schön gerechnet werden kann. Das Zweierkomplement wird gebildet, indem einfach alle Bits einer Zahl umgekehrt werden, und am Schluss eine 1 addiert wird.

Als Beispiel nehmen wir die Zahl 5: 0101. Wir bilden das Zweierkomplement um die Zahl –5 zu erhalten. Zuerst alle Bits umkehren: 0101 => 1010. Danach 0001 addieren: 1011. Die Abb. 1.18a) zeigt alle 4-Bit breiten Zahlenwerte im Zweierkomplement. Das detektieren von Overflows (wenn ein Resultat zu gross ist, um mit der vorhandenen Bitbreite dargestellt zu werden) wird dadurch aber einiges schwieriger und wird hier nicht weiter besprochen.

Aufgabe 1.9: Berechnen Sie mit dem Zweierkomplement die Ergebnisse für: - 5 - 2, 7 – 8, -8 + 4 und –5 + 7.

Wir benutzen also für die Subtraktion zweier Operanden eine Addition und setzen aber den zweiten Operand in sein Zweierkomplement. Dafür müssen wir jedes Bit des zweiten Operanden umkehren, was in unserer 1- Bit ALU durch einen zusätzlichen Multiplexer kein Problem ist. Die Kosten sind ein drittes Kontrollsignal für die ALU. In Abb. 1.18b) sind die neuen Komponenten in unserer 1-Bit ALU schwarz hervorgehoben.

a)

Dezimalwert	Zweierkomplement
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

b)

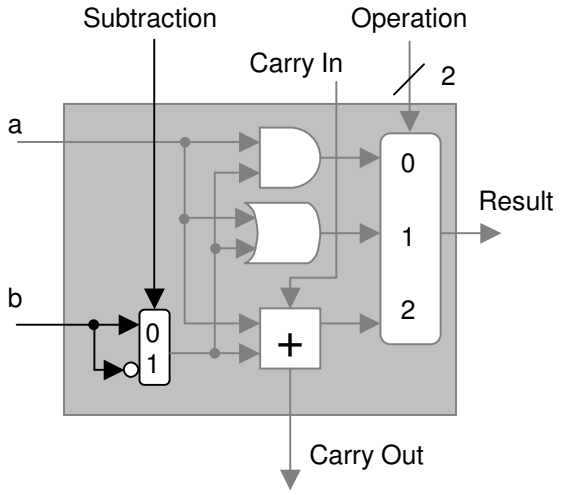


Abb. 1.18: In a) ist eine Tabelle von positiven und negativen Dezimalwerten zu sehen, die in der rechten Spalte in ihrer 4-Bit breiten Zweierkomplement-Darstellung abgebildet sind. In b) wird die 1-Bit ALU um einen Multiplexer für die Subtraktionsunterstützung erweitert. Die ALU kann dadurch das Bit des Operanden b umkehren, hat nun aber auch ein drittes Kontrollsignal „Subtraction".

Für die Vervollständigung des Zweierkomplements muss in der ALU aber noch eine 1 addiert werden. Da bei der Additionsoperation das „Carry In" beim Least Significant Bit immer mit 0 gespiesen wird, können wir diesen Eingang bei der Subtraktionsoperation dazu benutzen, noch eine 1 hinzuzufügen. Wir verknüpfen also wie in Abb. 1.19 das Subtraktionskontrollsignal mit dem „Carry In" der 1-Bit ALU [0].

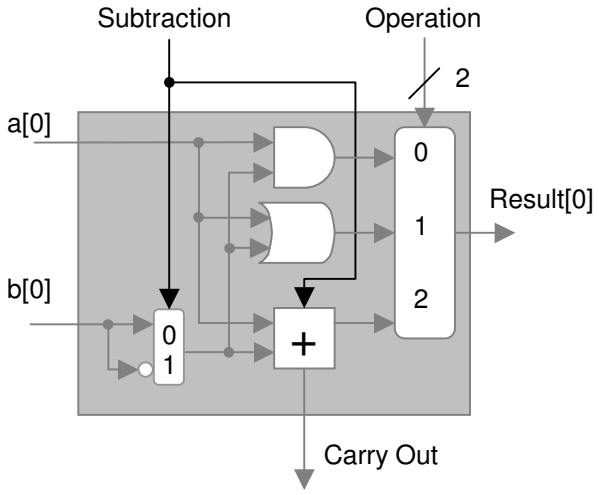
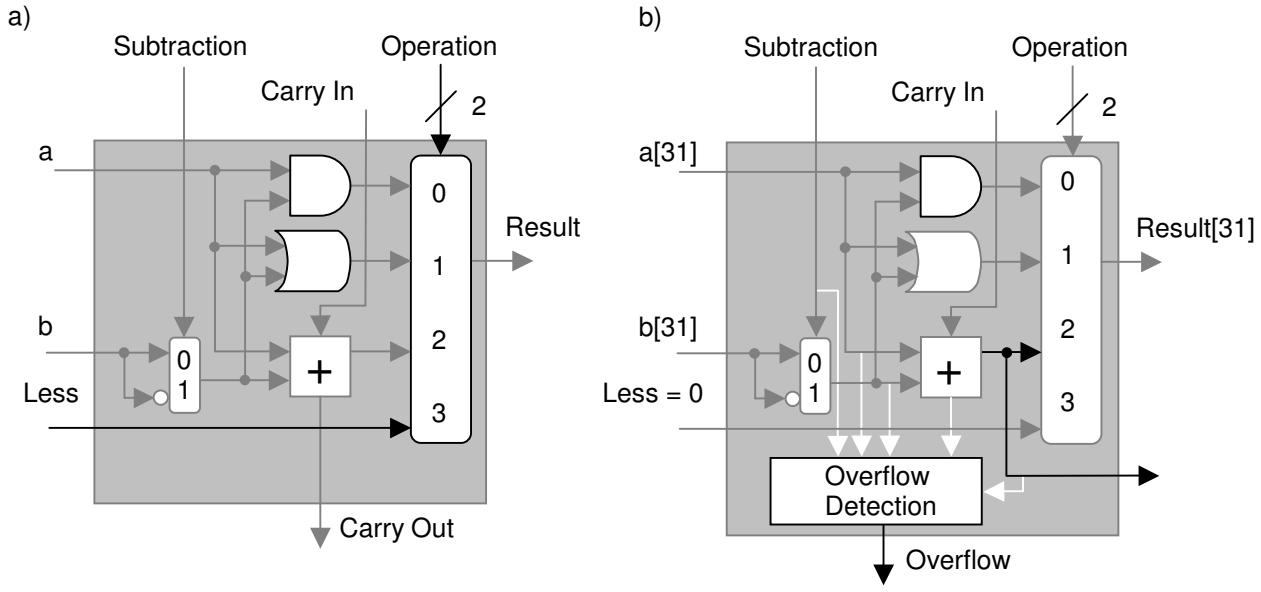


Abb. 1.19: Das Kontrollsignal „Subtraction“ wird bei der 1-Bit ALU des Least Significant Bits mit dem „Carry In“-Eingang verbunden, damit bei einer Subtraktion nicht nur die Bits umgekehrt werden, sondern auch noch eine 1 addiert wird. Dadurch wird das Zweierkomplement vollständig gebildet.

Nun benötigen wir noch Operationen um 2 Operanden zu vergleichen. Zuerst möchten wir auswerten ob a < b ist. Im Resultat soll eine 1 ausgegeben werden falls a < b zutrifft, sonst soll 0 ausgegeben werden. Diese Operation nennen wir “set on less than”. Der Vergleich ist gleichbedeutend wie a – b < 0. Wir können also die Operanden subtrahieren und dann das Most Significant Bit des Subtraktionsresultates in das Least Significant Bit des eigentlichen „set on less than“-Resultates schreiben. Das Most Significant Bit des Subtraktionsresultates ist ja das Vorzeichen des Wertes, positiv oder negativ, und zeigt uns somit an ob das Ergebnis größergleich oder kleiner 0 ist. In Abb. 1.20b) wird das Most Significant Bit aus dem Subtraktionsresultat abgezweigt. Wir müssen aber noch allen 1-Bit ALU-Komponenten angeben, dass das Resultat von der Vergleichsoperation ausgegeben werden soll. Hierfür erweitern wir unseren 3 Weg-Multiplexer der 1-Bit ALU Komponente zu einem 4-Weg Multiplexer wie in Abb. 1.20a). Das abgezweigte Most Significant Bit wird nun mit dem Eingang „Less“ der 1-Bit ALU Komponente mit Index 0, das Least Significant Bit des Resultates verbunden. Alle anderen Bits des „set on less than“ -Resultates sind immer 0, und deshalb sind auch alle anderen „Less“-Eingänge mit 0 gespiesen, wie es in Abb. 1.20c) zu sehen ist.



c)

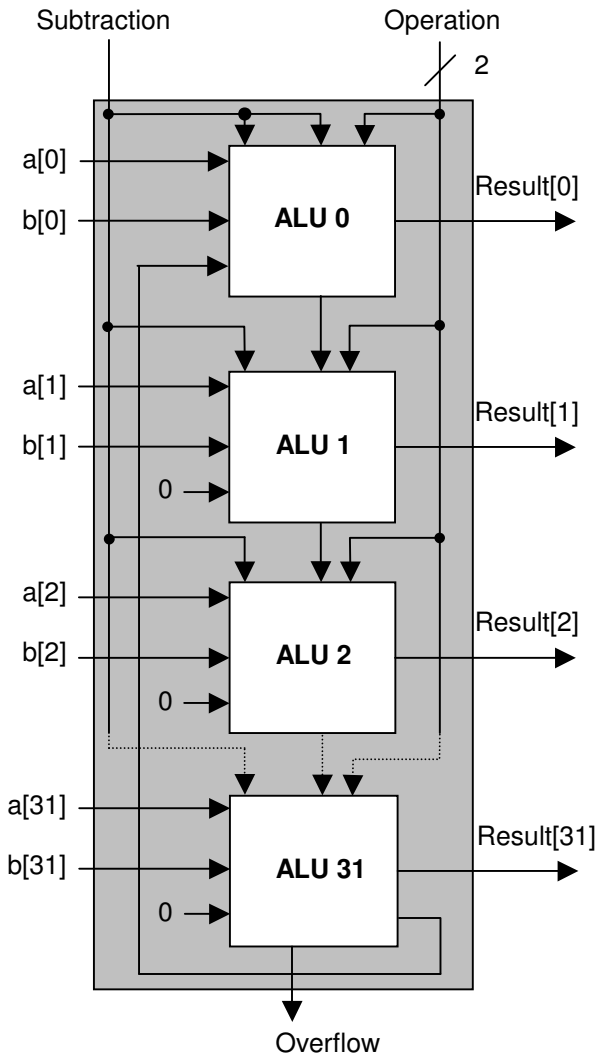
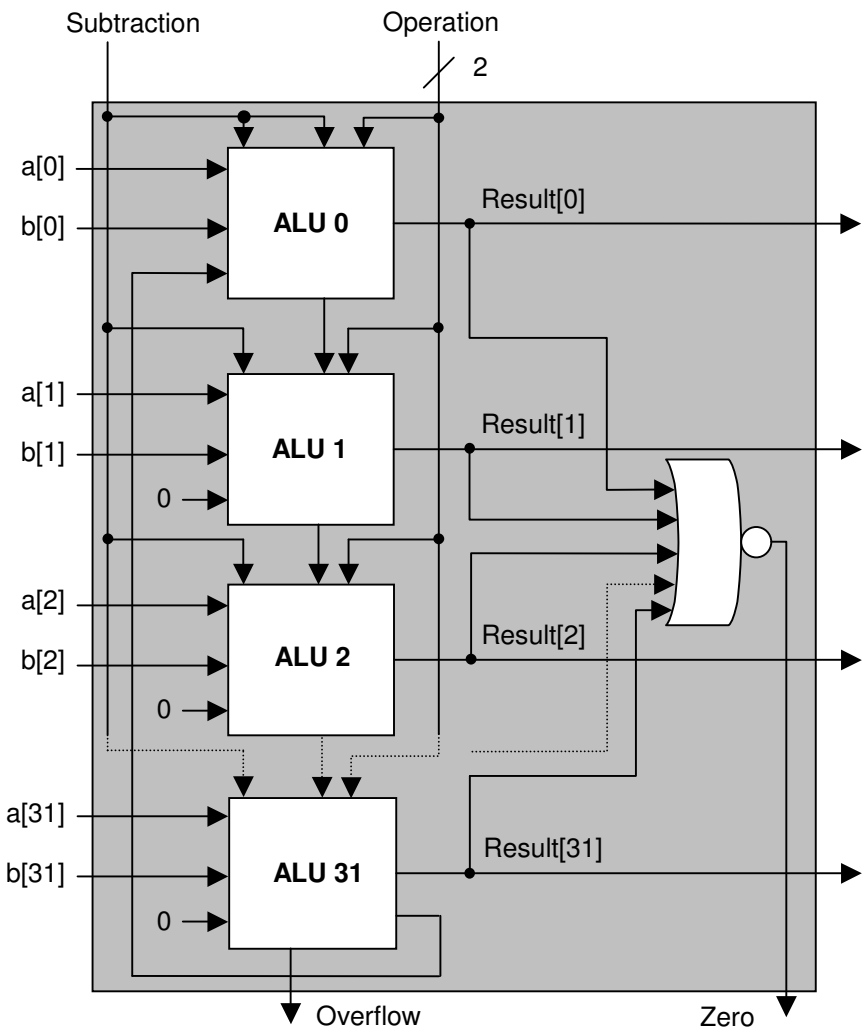


Abb. 1.20: In a) ist die Erweiterung der 1-Bit ALU Komponente um die Resultatsausgabe „set on less than“ abgebildet. In b) wird die 1-bit ALU Komponente des Most Significant Bit gezeigt. In ihr wird das Resultat aus der Addition abgezweigt und in die 1-Bit ALU Komponente des Least Significant Bits geleitet. Alle anderen Less-Eingänge der ALU Komponenten sind 0. Es ist auch eine kompliziertere Overflow-Detection Schaltung in der Most Significant 1-Bit ALU nötig, auf diese wird aber nicht weiter eingegangen. In c) werden die 1-Bit ALU Komponenten zu einer 32-Bit ALU mit den Operationen AND, OR, Addition, Subtraktion und „set on less than“ zusammengesetzt.

Zum Abschluss wollen wir die ALU noch mit einem Output-Signal erweitern, das uns immer anzeigt, ob das Resultat gleich 0 ist. Dazu können einfach alle Signalleitungen des Resultates mit einem Mehrweg-OR Gatter verknüpft, und das Signal danach umkehrt werden. Dadurch erhalten wir eine 1 auf dieser Ausgangsleitung wenn das Resultat gleich 0 ist. Abb. 1.21 zeigt die fertige ALU und das Schaltsymbol dazu.

a)



b)

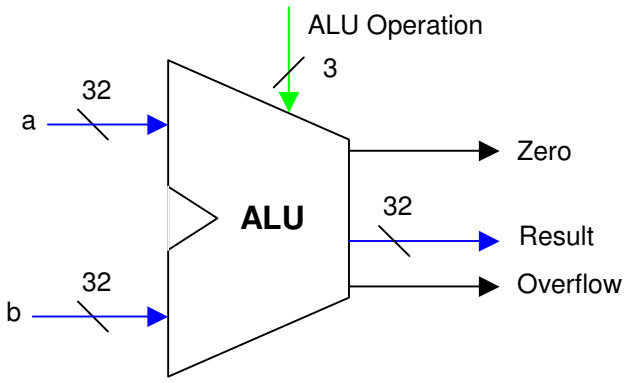


Abb. 1.21: In a) ist die komplette Schaltung der 32-Bit ALU angegeben. In b) ist das Schaltungssymbol der 32-Bit ALU dargestellt. Die Kontrollsignale „Operation“ und „Subtraction“ wurden zu einem 3-Bit breiten Kontrollsignal „ALU Operation“ zusammengefasst.

Wir wollen nochmals kurz die Funktionalitäten unserer ALU zusammenfassen. Neben den zwei 32-Bit Operanden a und b haben wir eine Kontrollleitung die die Subtraktion steuert. Zwei Kontrollleitungen bestimmen welches Resultat der Operationen AND, OR, Addition und „set on less than“ ausgegeben werden soll.

Funktion	Operation	Subtraction
AND	00	0
OR	01	0
Addition	10	0
Subtraktion	10	1
Set on less than	11	1

Abb. 1.22: Tabelle mit den ALU-Operationen. Die zwei rechten Spalten sind die ALU Kontrollsignale mit den Werten, die bei einer gewünschten Operationsausführung gesetzt werden müssen.

Aufgabe 1.10: Zeichne die vollständige Schaltung einer 4-Bit ALU. Auf den Inputs a und b liegen die Werte 2: 0010 und 5: 0101 an. Beschrifte sämtliche Leitungen der ALU. Was ist das Resultat für die Operationen AND, OR, Addition, Subtraktion und „set on less than“?

1.3 Speicherbausteine

Als nächstes betrachten wir Speicherbausteine. Wir interessieren uns auch hier weniger für den physikalischen Aufbau, sondern mehr für die Funktionsweise solcher Komponenten. In einen Speicherbaustein kann durch eine Write-Operation ein 32 Bit breiter Wert gespeichert werden. Durch eine Read-Operation kann der Wert zum gewünschten Zeitpunkt wieder aus dem Speicher ausgelesen werden.

Allen Speicherbausteinen ist gemeinsam, dass sie ein regelmässiges Signal benötigen um zu entscheiden, wann eine nächste Write-Operation ausgeführt werden soll. Dieses Signal nennt sich die „Clock“. Jeder Baustein kann in einem Clock-Zyklus eine Read oder/und eine Write-Operation ausführen. Auch wenn es jetzt allen Elektrotechnikern die Haare im Nacken aufstellt: wir stellen uns die Clock idealisiert vor und vernachlässigen sie wo immer möglich. Idealisiert bedeutet, das Clock-Signal ist absolut regelmässig und kommt bei allen Speicherkomponenten zum selben Zeitpunkt an.

Als erster Baustein betrachten wir ein einzelnes **Register**. Das Schaltsymbol ist in Abb. 1.23a) aufgezeigt.

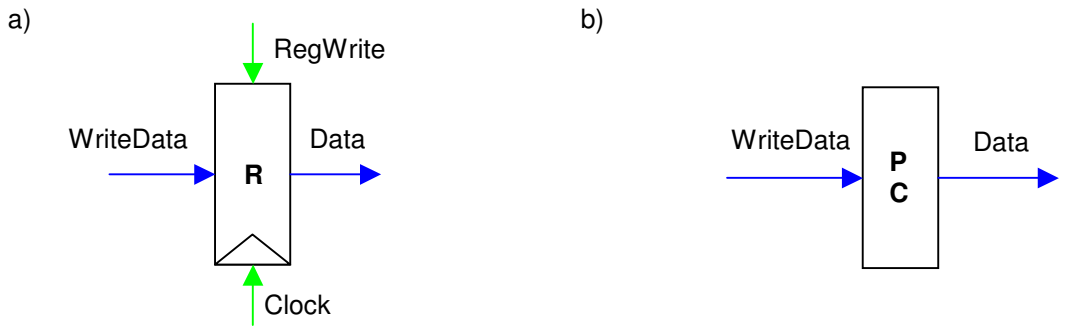


Abb. 1.23: In a) ist das Schaltungssymbol für ein normales Register abgebildet. In b) wird das spezielle Register „Program-Counter“ dargestellt. Der Eingang für das Clocksignal wird vernachlässigt. Das Kontrollsignal RegWrite ist beim Program-Counter immer 1 und wird deshalb weggelassen.

Am „Data“ Ausgang des Registers kann jederzeit der gespeicherte 32-Bit Wert aus dem Register ausgelesen werden. Ist das Kontrollsignal RegWrite auf 1, wird beim nächsten Clock-Signal der Wert am Eingang „WriteData“ in das Register geschrieben. Dadurch kann in einem Takt (Clock-Zyklus) ein Wert gelesen und geschrieben werden.

Für unseren Rechner benötigen wir ein spezielle Form eines einzelnen Registers, der sogenannte **Program-Counter**. In diesem Register wird bei jedem Takt-Zyklus ein neuer Wert eingeschrieben, das Kontrollsignal RegWrite ist also immer auf 1 gesetzt. Im Schaltungssymbol, Abb. 1.23b) wird der RegWrite Eingang und die Clock nicht eingezeichnet. Der Wert in diesem Register ist die Speicher-Adresse im Instruction Memory, wo die nächste Instruktion für die ALU gespeichert ist. Zum Instruction Memory kommen wir in Kürze, zuerst wollen wir uns aber noch den Register-Block der ALU, wie er in Abb. 1.24 dargestellt ist, betrachten.

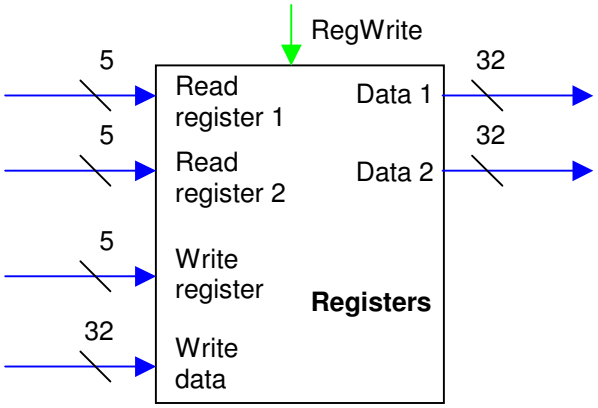


Abb. 1.24: Zwei 5-Bit Register Adressen an „Read register“ Eingängen veranlassen den Block zwei 32-Bit Datenwerte aus den entsprechenden Registern auszugeben. Ist das Kontrollsignal RegWrite auf 1, so wird beim nächsten Clockschlag der Wert am „Write data“ Eingang in das Register mit der Adresse „Write register“ geschrieben.

Der **Register-Block** umfasst 32 einzelne Register in denen die Operanden für ALU-Operationen gespeichert sind. In einem Zyklus können zwei Register an den Ausgängen Data 1 und 2 gelesen werden, z.B. die Operanden für die ALU Subtraktion. Das Resultat aus der ALU wird beim nächsten Clockschlag wieder in ein drittes Register des Blocks geschrieben. Hierfür wird der Eingang „Write Data“ sowie das Kontrollsignal „RegWrite“ benötigt. Um noch zu bestimmen welche zwei Register von den 32 gelesen werden sollen, und in welches Register allenfalls ein Resultat geschrieben werden soll, benötigt man noch 3 jeweils 5 Bit breite Adressesignale. (2 hoch 5 = 32).

Aufgabe 1.11: Bilden Sie aus einzelnen Registern sowie aus logischen Gattern und Multiplexern die Schaltung für den Register-Block.

Die 32 Register sind je einem bestimmten Zweck zugeordnet und dienen hauptsächlich dem Handling von Prozeduren und dem Stack-Management im Data Memory. Wir hören im Kapitel 2 einiges mehr darüber. In Abb. 1.25 ist eine Tabelle mit unserer Konvention über den Registergebrauch abgebildet.

Name	Index	Gebrauch	Auf Stack
$zero	0	Beinhaltet immer 0	Nein
$at	1	Vom Linker verwendet	Nein
$v0-$v1	2-3	Prozedur-Resultate	Nein
$a0-$a3	4-7	Prozedur-Argumente	Nein
$t0-$t7	8-15	Für temporäre Zwischenwerte	Nein
$s0-$s7	16-23	Prozedurvariablen	Ja
$Hi-$Lo	24-25	unterstützt Multiplikation und Division	Nein
$k0-$k1	26-27	Vom Betriebsystem verwendet	Nein
$gp	28	Pointer auf globale Variablen	Ja
$sp	29	Stackpointer	Ja
$fp	30	Framepointer	Ja
$ra	31	Prozedur-Rücksprungadresse	Ja

Abb. 1.25: Tabelle mit der Konvention wie die Register genutzt werden. In der Spalte „Name“ sind die Eigennamen für Register angegeben. Die Register können aber immer auch mit $r[Registernummer] adressiert werden. Die Registernummer ist in der Spalte „Index“ aufgeführt. In der Spalte „Gebrauch“ ist die Konvention angegeben, für was oder von wem ein Register benutzt wird. Die Spalte „Auf Stack“ gibt an, ob bei einem Prozedurwechsel die Register auf den Stack ins Data Memory geschrieben werden müssen.

Nun wollen wir den Hauptspeicher betrachten. Der Hauptspeicher, oder auch Memory, nimmt eine 32-Bit Adresse auf und gibt den Inhalt des Speichers an dieser Adresse zurück. Mit einer Adresse wird ein einzelnes Byte (8 Bit) adressiert. Es sind also 2 hoch 32 Byte (4 GigaByte) adressierbar. Da eine Adresse ein Byte adressiert, wir aber als Resultat sowieso immer 32-Bit Wörter (also 4 Bytes) zurückbekommen, können die zwei hintersten Bits der Input-Adresse auf 0 gesetzt werden. Dadurch wird die nächst tiefere, durch 4 teilbare Adresse angesprochen. Man sagt: „Der Speicher hat ein 4 Alignment“. Ein normales Memory unterstützt aber auch das speichern und laden von einzelnen Bytes.

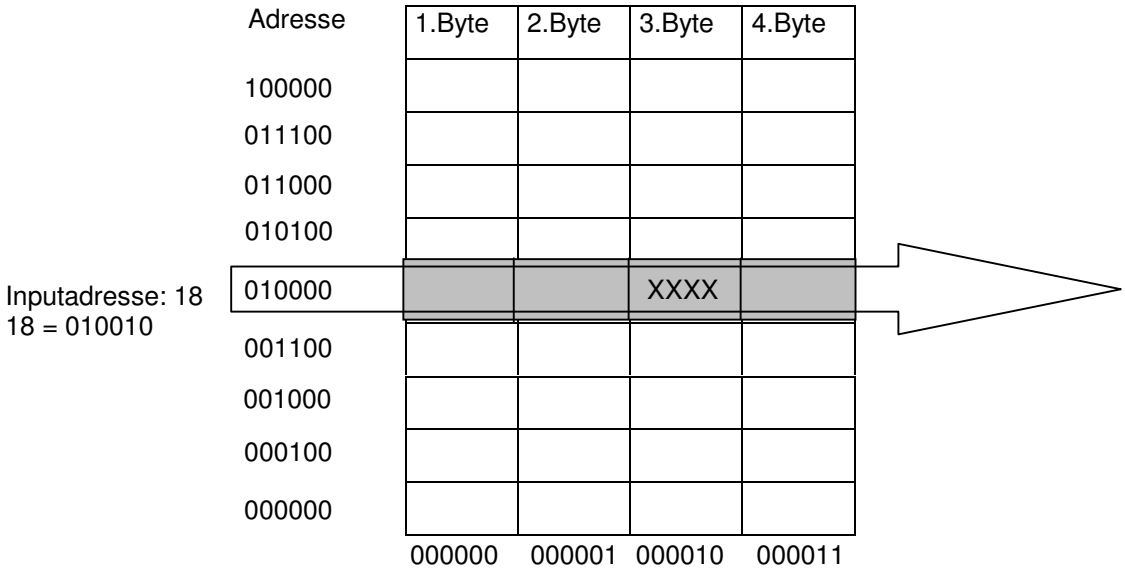


Abb. 1.26: Um das gewünschtes Byte an der Adresse 18 zu lesen, werden 4 Byte ab der Adresse 16 ausgelesen. Dies, weil der Speicher immer 4 Bytes als Resultat zurückgibt. Der Speicher hat ein 4-Alignment.

Wir wollen zwei verschiedene Arten von Memory unterscheiden. Das Instruction Memory und das Data Memory.

Im **Instruction Memory** wird der Maschinencode eines Programms, eine Folge von 32-Bit breiten Instruktionssignalen an den Rechner, gespeichert. Wie wir aus einem Assembler-Programm einen Maschinencode erzeugen wird im Kapitel 3 besprochen. Da sich die Instruktionen während des Programmablaufes nicht ändern und bei jedem Clock-Zyklus eine neue Instruktion aus dem Instruction Memory ausgelesen wird, wird nur ein Eingang für die Adresse sowie der Ausgang um die Instruktion auszulesen benötigt. In Abb. 1.27b) ist der Instruction Memory Eingang für die Clock wieder vernachlässigt worden.

Nun gibt es noch das **Data Memory**. In ihm werden die Daten gehalten, die mit den Maschinencode-Instruktionen aus dem Instruction Memory bearbeitet werden. Dies können der Stack, globale Konstanten und Variablen, aber auch ganze Datenstrukturen sein. Mehr dazu im Kapitel 2. In einem Clock Zyklus kann entweder eine Speicheradresse mit einem 32-Bit Wert gefüllt werden (wieder die 4 Bytes), oder es kann ein Wert aus einer bestimmten Adresse ausgelesen werden. Beachten Sie, dass sie mit der Adressierung einzelne Bytes schreiben oder lesen könnten, wir werden aber wegen dem Alignment immer nur 4 Bytes auf einmal manipulieren. Das Data Memory benötigt wie in Abb. 1.27a) ein Eingangssignal für die Adresse. Zwei verschiedene Kontrollsignale zeigen an, ob ein Wert bei dieser Data Memory Adresse am Ausgang „Data“ gelesen werden soll (MemRead = 1), oder ob der Wert beim „Write Data“ Eingang in diese Speicheradresse geschrieben werden soll (MemWrite = 1). MemWrite und MemRead können nicht gleichzeitig auf 1 gesetzt sein. Sind beide Kontrollsignale 0, so passiert nichts.

Wie der Maschinencode ins Instruction Memory gelangt und die zu bearbeitenden Daten ins Data Memory gelangen wird hier nicht weiter besprochen. Man kann sich aber z.B. vorstellen, das beim Starten des Computers zuerst eine Lochkarte mit dem Maschinencode in das Instruction Memory und eine Lochkarte mit den zu bearbeitenden Daten in das Data Memory eingelesen wird.

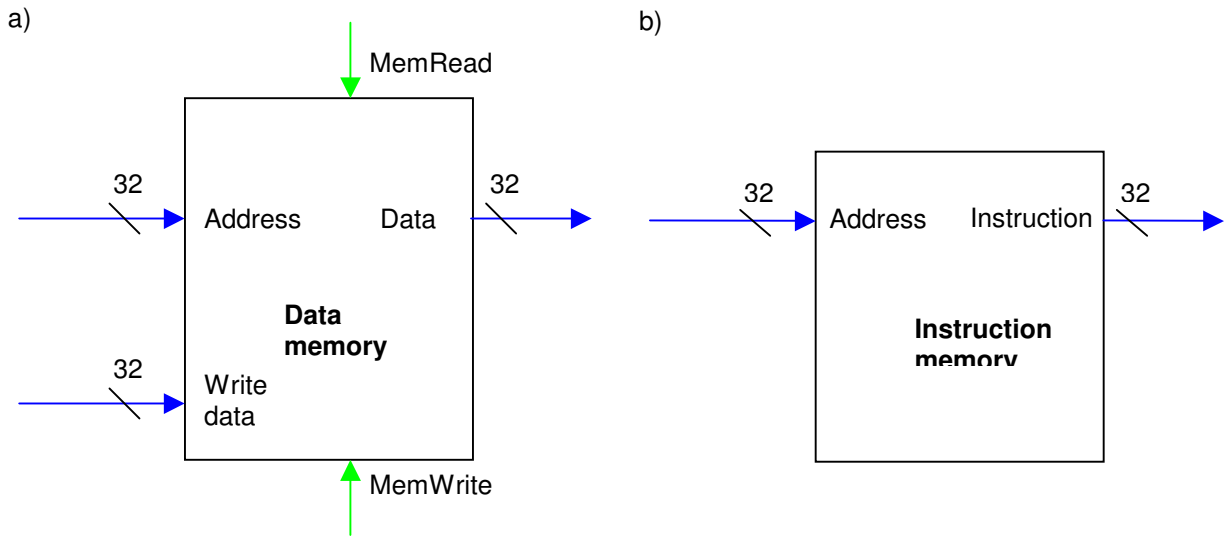


Abb. 1.27: In a) ist das Schaltungssymbol für ein Data Memory abgebildet. In b) ist das Symbol für das Instruction Memory aufgeführt. Der Eingang für die Clock ist bei beiden Schaltungssymbolen vernachlässigt worden.

Aufgabe 1.12:

a) Warum wurden für das Instruction Memory und das Data Memory zwei verschiedene Speicherbausteine verwendet, und nicht einfach zwei verschiedene Adressbereiche im selben Memory reserviert?

b) Zeichnen Sie ein Data Memory. Welche Eingangsleitungen werden mit welchen Werten belegt, falls
   1. der Wert 34 in der Memory-Adresse 84 gespeichert werden soll ?
   2. ein Wert aus der Memory-Adresse 104 geladen werden soll?

c) Was wäre bei einer realen Clock noch zu berücksichtigen?

1.4 Zwei Beispielschaltungen

Zum Abschluss des Kapitels wollen wir noch zwei erste nützliche Schaltungen für den Rechner besprechen. Als erstes bauen wir eine Schaltung, die sequentielle 32-Bit Instruktionen der Reihe nach aus dem Instruction Memory ausliest.

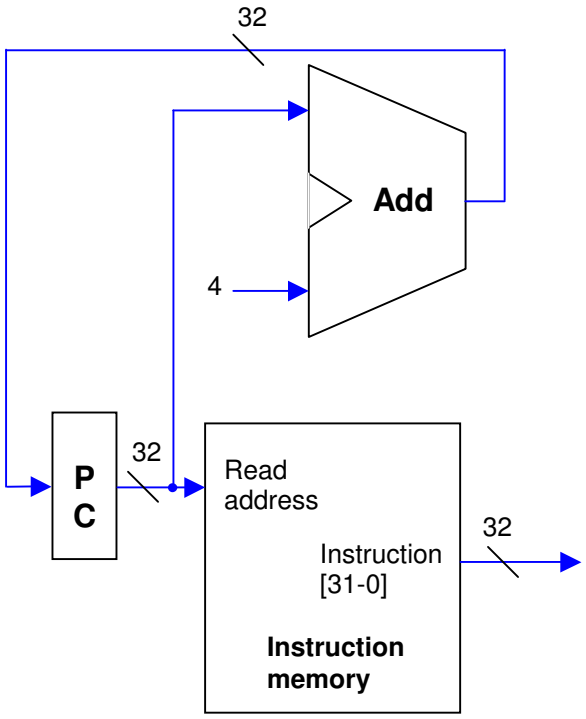


Abb. 1.28: Ein Program-Counter wird mit einem Addierer und dem Instruktion Memory verknüpft. Bei jedem Taktschlag wird eine neue Adresse im Program-Counter gespeichert, da diese in jedem Zyklus durch den Addierer um 4 erhöht wird. Bei jeder neuen Adresse wird eine neue 32-Bit breite Instruktion aus dem Instruction Memory ausgegeben.

Die Schaltung hat zu Beginn den Wert 0 im Program-Counter gespeichert. Dabei wird im Instruction Memory die Instruktion an dieser Adresse 0 ausgelesen. Wir haben ja angenommen, dass sich die Instruktionen bereits im Instruction Memory befinden. Gleichzeitig wird der Wert des Program-Counters im Addierer mit 4 addiert. Dies entspricht der Adresse der nächsten 4 Bytes oder 32 Bits im Instruction Memory. Das Resultat wird beim nächsten Clockschlag wieder in den Program-Counter geschrieben, der das Instruction Memory wiederum dazu veranlasst die Instruktion an der Adresse 4 auszugeben. Die Instruktionsadresse wird im Addierer wieder erhöht und so weiter und so fort. Der Endeffekt ist, das bei jedem Clockschlag die nächste seriell angegebene 32-Bit Maschinencode-Instruktion aus dem Instruction Memory ausgelesen wird.

Als zweite Schaltung kombinieren wir den Register-Block mit der ALU.

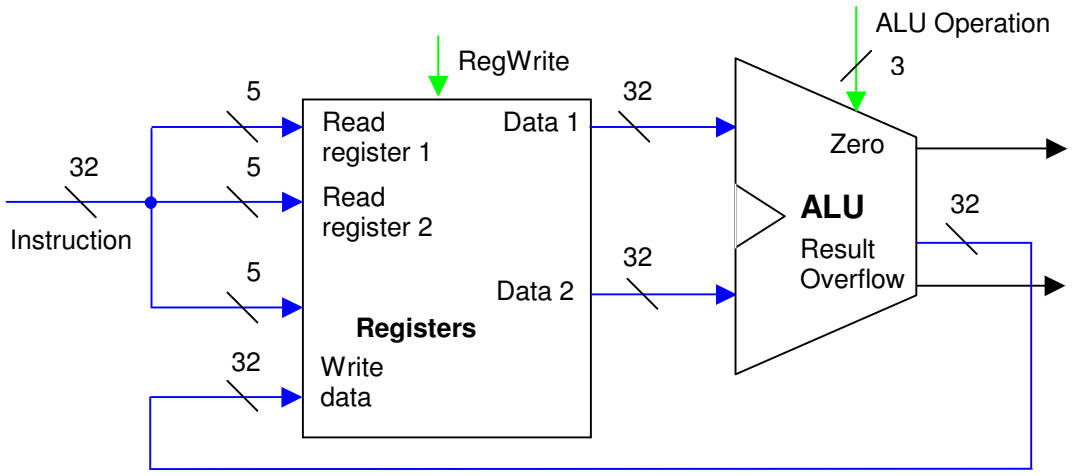


Abb. 1.29: Eine Schaltung kombiniert den Register-Block mit der ALU.

Damit soll zum Abschluss des Kapitels ein erster Eindruck gewonnen werden, was uns im Kapitel 3 erwarten wird. Wir haben eine 32 Bit breite Instruktion aus dem Instruction Memory kommend. Wollen wir eine Addition, Subtraktion, OR oder AND Operation auf zwei in Register geladene Operanden ausführen, so werden 15 Bits der 32 Instruktionsbits verwendet, um die zwei Registeradressen für die Operanden sowie die Adresse des Registers für das Resultat anzugeben. Es bleiben noch 17 Bits mit denen die Kontrollsignale für den Rechner angegeben werden können. Wir werden nur 6 Bits dieser verbleibenden 17 Bits für die Steuerung der Kontrollsignale benötigen: den sogenannten Opcode. Der Opcode der Instruktion ist ein Teil der Instruktion, der die auszuführende Operation definiert und die benötigen Kontrollsignale an den Rechner bestimmt. Wir bilden den vollständigen Instruktionscode (Maschinencode) inklusiv Opcode in Kapitel 3.

Zusammenfassung

Im ersten Abschnitt des Kapitels haben wir die Darstellung von komplexen Signalleitern aufgezeigt, sowie aus den Leitern eine Shift left 2 und eine Sign-Extension Komponente gebaut.

Der zweite Abschnitt hat mit einer Wiederholung von logischen Gattern begonnen. Der Multiplexer diente als nützliches Aufwärmen, bevor wir aus 32 parallel verknüpften 1-Bit Addierer-Schaltungen einen 32-Bit Addierer zusammensetzten. Danach haben wir die 1-Bit Addierer Schaltung um logisches AND und OR erweitert und eine 1-Bit ALU Schaltung erhalten. Wieder konnten wir durch deren parallele Kombination eine erste 32-Bit ALU erstellen. Für die Erweiterung der 32-bit ALU um die Subtraktionsoperation wurde das Zweierkomplement zu binären Zahlen repetiert und danach die Schritte gezeigt, wie in der ALU ein Operand ins Zweierkomplement gesetzt wird. Für die Subtraktion wurde aber auch ein weiteres Kontrollsignal benötigt. Bei Vergleichsoperation „set on less than“ wird eine Subtraktion ausgeführt und ein Resultat-Bit umgeleitet, so das wir bei a < b eine 1 als Resultat erhalten, bei a >= b eine 0. Das Ausgangs-Kontrollsignal Zero welches uns immer anzeigt ob ein Resultat gleich 0 ist, vervollständigte die 32-Bit ALU.

Etwas abstrakter haben wir die Funktionalität von Registern, insbesondere dem Program-Counter und dem Register-Block, betrachtet. Hier galt es hauptsächlich die Ein- und Ausgangsleitungen zu spezifizieren. Für die vielen Anwendungs- und Instruktionsdaten haben wir danach die zwei spezialisierten Speicherbausteine Instruction Memory und Data Memory betrachtet. Dabei haben wir die Adressberechnung für einzelne Bytes vorgestellt, die aber durch ein 4-Alignment im Speicher immer zu

durch 4 teilbare Adressen auswerten und immer 4 Byte grosse Werte manipulieren. Das Clock-Signal als Taktgeber für die Speicherbausteine haben wir nur minimal angesprochen, aber mit der Hoffnung, dass sie vom Leser im Hinterkopf behalten wird.

Eine Schaltung zum Auslesen der Maschinencode-Instruktionen aus dem Instruction Memory, sowie eine Schaltung die den Register-Block mit der ALU verbindet, sollten zum Kapitelabschluss die Neugierde auf mehr „Rechnerstruktur“ wecken.

Lernziele

Haben Sie die folgenden Lernziele des Kapitels erfüllt? Falls Sie sich nicht ganz sicher fühlen, schlagen Sie die Punkte nochmals nach. Fühlen Sie sich beim Umgang mit dem Stoff sicher? Dann auf, auf zum Kapiteltest.

- Sie kennen die Darstellungsweise von einfachen und komplexeren Datenleitungen
- Sie kennen die Wirkung eines Shift left 2 und einer Sign-Extension
- Sie wissen wie ein Multiplexer für einfache und komplexere Signale wirkt und aufgebaut ist
- Sie wissen, wie die Schaltung für einen Addierer aufgebaut ist
- Sie kennen die Operationen der entworfenen ALU und wissen, wie die Schaltung aufgebaut ist
- Sie wissen was ein Register-Block beinhaltet und welche Signaleingänge für ein gewünschtes Ergebnis gesetzt werden müssen
- Sie wissen wie im Memory auf eine Speicheradresse zugegriffen werden kann um einen Wert zu lesen oder zu schreiben

Kapiteltest 1

Aufgabe 1:

Zeichnen Sie einen Multiplexer, der zwischen 4 2-Bit breiten Eingangssignalen eines auswählen kann. Erstellen Sie dazu eine Tabelle für die benötigten Kontrollsignal-Werte, die zur Auswahl eines der 4 Eingangssignale benötigt werden.

Aufgabe 2:

a)
Bilden Sie das Zweierkomplement zu folgenden Binärwerten:

- 001101010
- 101011001

b)
Berechnen Sie die Ergebnisse für folgende Rechenoperationen. Rechnen Sie dabei mit binären Zahlenwerten. Die binären Zahlen sind dabei maximal 4 Bit breit:

- -3 + 5
- 5 + 4
- 2 - 8

Aufgabe 3:

Zeichnen Sie eine vollständige 2-Bit breite ALU. Welche Werte haben die Ausgangssignale „Result“ und „Zero“ bei jeder Operation, falls die Eingangssignale a und b den Wert 01 leiten.

Aufgabe 4:

Zeichnen Sie ein Daten Memory. Welche Signale werden mit welchen Werten belegt, falls

a) der Wert 56 in der Memory-Adresse 236 gespeichert werden soll ?
b) der Wert an der Memory-Adresse 464 ausgegeben werden soll ?

Kapitel 2: Assembler

Einleitung

In diesem Kapitel wollen wir einen eigenen Assembler Programmierbefehlssatz zusammenstellen sowie die gängigsten Programm-Verzweigungstechniken wiederholen. Für Prozeduraufrufe werden wir auch das Stack-Management nochmals etwas beleuchten, es wird aber keine vollständige Abhandlung über diese Problematik wiedergegeben. Nach einer kurzen Einführung wird das Kapitel in folgende 3 Abschnitte gegliedert:

1 Im ersten Abschnitt werden wir die einzelnen Befehle unserer Assembler-Programmiersprache vorstellen und ihre genaue Wirkung erläutern. Wir werden die Befehle, die gleiche Parameter erwarten, in 3 verschiedene Formatklassen einteilen.

2 Der zweite Abschnitt bietet eine Repetition der häufigsten Techniken, um in Assembler den Programmfluss zu steuern. Konkret: wir betrachten zuerst einfache Programm-Anweisungen auf Konstanten, Register und Memory Variablen. Die Indexierung eines Datenarrays wird an einem Beispiel erläutert und danach wird dann eine „If-then-else“- Verzweigung und ein „while“Loop“-Anweisung in Assembler implementiert.

3 Der letzte Abschnitt zeigt auf, wie Prozeduren in Assembler realisiert werden. Dabei wird aufgezeigt, wie der Stack aufgebaut wird, und welche Stack- und Registermanipulationen nötig werden.

Einführung

Assembler ist eine Low-Level Programmiersprache, d.h. es wird mit Befehlen programmiert, die ein Rechner durch wenige entsprechende Operationsinstruktionen ausführen kann. Es sind also Befehle, die mehr oder weniger direkt in eine Maschinencode-Instruktion übersetzt werden können. Der programmierte Assemblercode ist aber nicht unbedingt von der Rechnerstruktur abhängig, obwohl die Assemblersprachen für die jeweiligen Rechnerstrukturen zugeschnitten sind. Unser Rechner wird z.B. mit der bereits in Kapitel 1 vorgestellten Registerkonvention arbeiten, und diese Registerkonvention wird auch direkt im Assemblercode benutzt. Dieser Code könnte aber auch für einen Rechner mit einer anderen Registerkonvention in Maschinencode umgewandelt werden. Eine Reihe von Assembler-Befehlen ist also noch etwas Unverständliches, Abstraktes für den Rechner. Erst wenn ein „Compiler“, ein Programm das Assemblerprogramme in den für die Maschine verständlichen Maschinencode übersetzt, können die Instruktionen im entsprechenden Rechner abgearbeitet werden. Im Kapitel 3 werden wir die Arbeit des Compilers übernehmen und Programme in einen Maschinencode übersetzen.

2.1 Assembler-Befehle

Als erstes betrachten wir Operationen, die zwei Register als Operanden nutzen und in einem dritten Register das Resultat aus der ALU zurückschreiben. In Abb. 2.1 ist als Beispiel der Assembler-Befehl für die Addition angegeben.

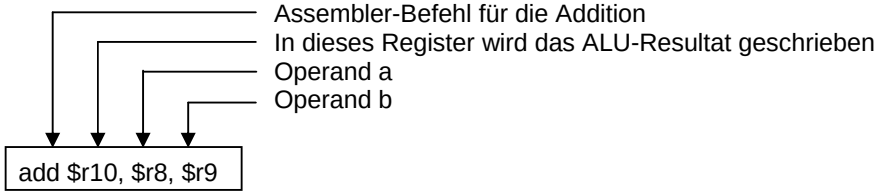


Abb. 2.1: Assemblerbefehl für die Addition des Wertes aus Register $r8 (=$t0) mit Register $r9 (=$t1). Das Resultat wird in das Register $r10(=$t2) geschrieben.

Die Assemblerbefehle für eine arithmetische Subtraktion, die logische AND und OR- Verknüpfung, aber auch die „set on less than“-Operation nehmen wie die Additionsoperation jeweils 2 Operanden aus Registern als Input und schreiben ein Resultat-Output von der ALU in ein drittes Register. Wir fassen all diese Befehle zu einer Klasse zusammen und taufen sie „R-Format“ Klasse. Das R steht für „Registeroperationen“. In Abb. 2.2 sind die Befehle der R-Format Klasse zusammengefasst.

Klasse R-Format

Operation	Assembler-Befehl	Wirkung
Addition	add $c, $a, $b	$c = $a + $b
Subtraktion	sub $c, $a, $b	$c = $a – $b
Logisches AND	and $c, $a, $b	$c = $a & $b
Logisches OR	or $c,$a, $b	$c = $a | $b
Set on Less than	slt $c, $a, $b	if $a < $b then $c = 1 else $c = 0

Abb. 2.2: Tabelle der Operationen der Klasse „R-Format“. In der ersten Spalte sind die Operationen aufgeführt, die zweite Spalte beinhaltet den dazugehörigen Assembler-Befehl. Alle aufgeführten Befehle haben als Parameter 3 Registeradressen. Die letzte Spalte zeigt die Wirkung der Operation auf das Resultatregister $c.

Aufgabe 2.1: Kurzer Verständnis-Check:

a) Welche Eigenschaft muss ein Assembler-Befehl haben, damit er der Klasse R-Format zugewiesen wird?
b) Wie wird der Assembler-Befehl geschrieben, wenn man Register $t2 von Register $t1 subtrahieren, und das Resultat in Register $t0 speichern möchte?

Wir wollen nun noch die Möglichkeit haben, die ALU Operationen auch mit einem Register und einem konstanten Wert als Operanden auszuführen. Die „R-Format“ Klasse ist dafür ungeeignet, da sie als Operanden eine 5-Bit Registeradresse erwartet. Würden wir einen Operanden als Konstante interpretieren, so wären Konstanten nur vom Wert 0 bis 31 möglich. Wir definieren für die ALU-Operationen mit einer Konstante eine neue Klasse von Befehlen. Die „I-Format“-Klasse. Das I steht für immediate (=direkt, sofort). Die Befehle der Klasse „I-Format“ haben als ersten Parameter wieder ein Register für das Resultat und als Operanden ein Register und eine Konstante, die direkt aus der Instruktion ausgelesen wird. Den Befehlen in Assembler wird ein „i“ angehängt.

Klasse I-Format

Operation	Assembler-Befehl	Wirkung
Addition Im.	addi $c, $a, K	$c = $a + K
Subtraktion Im.	subi $c, $a, K	$c = $a – K
Logisches AND Im.	andi $c, $a, K	$c = $a & K
Logisches OR Im.	ori $c,$a, K	$c = $a | K
Set on Less than Im.	slti $c, $a, K	if $a < K then $c = 1 else $c = 0

Abb. 2.3: Tabelle der Operationen der Klasse „I-Format“. In der ersten Spalte sind die Operationen aufgeführt, die zweite Spalte beinhaltet den dazugehörigen Assembler-Befehl. Alle aufgeführten Befehle haben als Parameter 2 Registeradressen und eine Konstante K. Die letzte Spalte zeigt die Wirkung der Operation auf das Resultatregister $c.

Als nächstes benötigen wir Befehle um einen Wert bei einer bestimmten Memory-Adresse aus dem Data Memory in ein Register zu laden, oder einen Wert eines Registers im Memory bei einer gegebenen Adresse abzuspeichern. Hierfür gibt es 2 Befehle: LoadWord (lw) und StoreWord (sw). Bei diesen Befehlen wird aber nicht nur einfach eine Adresse aus der Instruktion ausgelesen und geladen, sondern es wird eine Basis-Adresse aus einem Register genommen, und eine Konstante als Offset dazuaddiert. Dies vereinfacht die Programmierung vor allem für das Handling von Datenstrukturen wie Arrays oder Records, bei uns wird die Konstante aber meistens 0 sein. Die genauen Berechnungen für Adressen und Offsets werden in Kapitel 3 angegangen.

Bei einer LoadWord Instruktion muss neben dem Register für die Basis-Adresse und dem konstanten Adress-Offset angegeben werden, in welches Zielregister der Wert aus dem Memory gespeichert werden soll. Total also 2 Register und eine Konstante.

Bei der StoreWord Instruktion haben wir auch wieder die Basis-Adresse aus einem Register und die Offset-Konstante für die Adressberechnung. Hinzu kommt hier das Register, das den im Memory zu speichernden Wert enthält. Auch hier sind wieder 2 Register und eine Konstante anzugeben.

Die LoadWord und StoreWord Befehle können deshalb der „I-Format" Klasse zugeteilt werden. Die Schreibweise der Assembler-Operanden ist aber etwas ungewohnt, wie in Abb. 2.4 zu sehen ist.

erweiterte Klasse I-Format

Operation	Assembler-Befehl	Wirkung
Addition Im.	addi $c, $a, K	$c = $a + K
Subtraktion Im.	subi $c, $a, K	$c = $a – K
Logisches AND Im.	andi $c, $a, K	$c = $a & K
Logisches OR Im.	ori $c,$a, K	$c = $a | K
Set on Less than Im.	slti $c, $a, K	if $a < K then $c = 1 else $c = 0
Load Word	lw $c, K($a)	$c = Mem[K + $a]
Store Word	sw $c, K($a)	Mem[K + $a] = $c

Abb. 2.4: Erweiterte Tabelle der Operationen der Klasse „I-Format". In der ersten Spalte sind die Operationen aufgeführt, die zweite Spalte beinhaltet den dazugehörigen Assembler-Befehl. Alle aufgeführten Befehle haben als Parameter 2 Registeradressen und eine Konstante K. Die letzte Spalte zeigt die Wirkung der Operation auf das Resultatregister $c oder das Memory.

Es fehlen noch bedingte und unbedingte Programmverzweigungsbefehle. Zuerst zu den bedingten Verzweigungen. Hier führen wir die Befehle „branch on equal" und „branch on not equal" ein.

Der Befehl „branch on equal", zu deutsch: „Verzweige bei Gleichheit", nimmt zur Auswertung der Gleichheitsbedingung zwei Register und subtrahiert diese. Wird nun am ALU-Ausgang „Zero" eine 1 ausgegeben, was uns anzeigt dass die Registerwerte identisch sind, dann wird auf eine Instruktionsadresse gesprungen, die aus einem dritten Parameter (eine Konstante) berechnet werden kann. Auch diese Adressberechnung wird im Kapitel 3 genauer besprochen. Der Assemblerbefehl für „branch on equal" lautet „beq".

Der Befehl „branch on not equal" funktioniert genau gleich aber mit dem Unterschied, dass auf die Adresskonstante gesprungen wird, wenn bei der ALU-Subtraktion eine 0 am „Zero" Ausgang resultiert. Dies ist der Fall, falls die Registerwerte nicht identisch sind. Der Assemblerbefehl für „branch on not equal" lautet „bne".

Auch diese 2 Befehle können der I-Format Klasse zugewiesen werden.

vollständige Klasse I-Format

Operation	Assembler-Befehl	Wirkung
Addition Im.	addi $c, $a, K	$c = $a + K
Subtraktion Im.	subi $c, $a, K	$c = $a – K
Logisches AND Im.	andi $c, $a, K	$c = $a & K
Logisches OR Im.	ori $c,$a, K	$c = $a | K
Set on Less than Im.	slti $c, $a, K	if $a < K then $c = 1 else $c = 0
Load Word	lw $c, K($a)	$c = Mem[K + $a]
Store Word	sw $c, K($a)	Mem[K + $a] = $c
Branch on equal	beq $a, $b, K	if $a == $b then PC = K
Branch on not equal	bne $a, $b, K	if $a <> $b then PC = K

Abb. 2.5: Vollständige Tabelle der Operationen der Klasse „I-Format". In der ersten Spalte sind die Operationen aufgeführt, die zweite Spalte beinhaltet den dazugehörigen Assembler-Befehl. Alle aufgeführten Befehle haben als Parameter 2 Registeradressen und eine Konstante K. Die letzte Spalte zeigt die Wirkung der Operation auf das Resultatregister $c, das Memory oder den Programm-Counter.

Aufgabe 2.2: Nochmals ein kurzer Verständnis-Check:

a) Welche Eigenschaft muss ein Assembler-Befehl haben, damit er der Klasse I-Format zugewiesen wird?
b) Wie wird der Assembler-Befehl geschrieben, wenn man den Wert aus einer Memory-Adresse in Register $t1 und der Offset-Konstante 64 ins Register $t0 laden möchte?

Nun kommen wir zu unbedingten Programmverzweigungen, also Sprungbefehle die in jedem Fall den Programm-Counter mit einer neuen Instruktionsadresse belegen. Diese Befehle benötigen nur einen einzigen Parameter nämlich eine Konstante mit der die neue Instruktionsadresse angegeben wird. Diese Konstante im Assemblerbefehl wird häufig mittels einem sogenannten „Label" übergeben, wir sehen in Kürze einige Beispiele dazu (in Kapitel 3 wird zusätzlich aufgezeigt, warum dies so gemacht wird). Wir bilden eine neue Klasse von Befehlen, die „J-Format" Klasse. Das J steht für „Jump" (Sprung). Der allgemeinste Assemblerbefehl für einen Sprung heißt auch gerade „jump" (j), wie in Abb. 2.6 zu sehen ist.

Es gibt nun noch einen spezielleren unbedingten Sprung-Befehl, der für Prozedurverzweigungen benötigt wird. Bei einem Prozeduraufruf muss nicht nur auf eine neue Instruktionsadresse gesprungen werden, sondern es muss auch der Wert des aktuellen Programm-Counter im Register $ra gespeichert werden. Mit diesem Registerwert in $ra kann aus der aufgerufenen Prozedur wieder zur vorherigen Prozedur zurückgesprungen werden. Diese zwei Aktionen werden mit einem einzigen Befehl, dem „Jump and Link" Befehl (jal) ausgeführt. Es wird aber nur der Parameter mit der Adresskonstante benötigt, da der Programm-Counter immer im selben Register $ra gespeichert wird. Der Befehl gehört also auch zur Klasse J-Format.

vollständige Klasse J-Format

Operation	Assembler-Befehl	Wirkung
Jump	j K	PC = K
Jump and Link	jal K	$ra = PC + 4, PC = K

Abb. 2.6: Vollständige Tabelle der Operationen der Klasse „J-Format". In der ersten Spalte sind die Operationen aufgeführt, die zweite Spalte beinhaltet den dazugehörigen Assembler-Befehl. Alle aufgeführten Befehle haben als Parameter eine Konstante K, die normalerweise durch ein Label angegeben wird. Die letzte Spalte zeigt die Auswirkung der Operation auf das Spezialregister $ra sowie den Programm-Counter.

Aufgabe 2.3: Der letzte kurze Verständnis-Check:

a) Welche Eigenschaft muss ein Assembler-Befehl haben, Sie ahnen es schon, damit er der Klasse J-Format zugewiesen wird?
b) Wie wird der Assembler-Befehl geschrieben, wenn man auf eine Prozedur-Adresse angezeigt durch ein Label „TARGET" springen möchte?

Um nun aus einer abgearbeiteten Prozedur zurückzuspringen, können wir die Adresse im Register $ra wieder in den Programm-Counter laden. Wir haben aber noch keinen Befehl, der einen unbedingten Rücksprung mit einem Register als Parameter ausführen kann. Ein neuer Befehl ist nötig, der Befehl „Jump Register" (jr). Dieser Befehl hat eine einzige Registeradresse als Argument, was zu keiner bisher gebildeten Klasse passt. Wir können aber den Befehl der „R-Format" Klasse zuweisen und einfach die erste der 3 Registeradressen als Sprung-Parameter benutzen. Die beiden weiteren Registeradressen werden ignoriert.

vollständige Klasse R-Format

Operation	Assembler-Befehl	Wirkung
Addition	add $c, $a, $b	$c = $a + $b
Subtraktion	sub $c, $a, $b	$c = $a – $b
Logisches AND	and $c, $a, $b	$c = $a & $b
Logisches OR	or $c,$a, $b	$c = $a | $b
Set on Less than	slt $c, $a, $b	if $a < $b then $c = 1 else $c = 0
Jump Register	jr $ra	PC = $ra

Abb. 2.7: Vollständige Tabelle der Operationen der Klasse „R-Format". In der ersten Spalte sind die Operationen aufgeführt, die zweite Spalte beinhaltet den dazugehörigen Assembler-Befehl. Alle aufgeführten Befehle haben als Argumente 3 Registeradressen. Der Befehl jr benutzt jedoch nur eines davon. Die letzte Spalte zeigt die Wirkung der Operation auf das Resultatregister oder den Programm-Counter.

Es gibt nun noch eine Reihe von indirekteren Befehlen, das heißt, Befehle die vom Compiler zuerst in ein oder zwei direkte Befehle umgewandelt werden. Als Beispiel wollen wir einen Befehl „Branch on greater than“ einführen (bg). Dieser Befehl führt eine bedingte Sprungverzweigung aus, falls a > b zutrifft. Dieser Befehl kann durch die Befehle slt und beq gebildet werden:

```
bg $t0, $t1, LOOP        //falls der Wert im Register $t0 grösser als der Wert im Register $t1
                         //ist, dann wird auf die Adresse des Labels „LOOP“ gesprungen
=

slt $t2, $t1, $t0        //falls $t0 grösser als $t1 ist wird in $t2 eine 1 abgelegt
bne $zero, $t2, LOOP     //falls $t2 nicht 0 ist wird auf die Adresse des Labels „LOOP“ verzweigt
```

Ein weiterer sehr bekannter, indirekter Befehl ist das Kommando „move“. Mit move kann eine Konstante in ein Register geladen werden. Der gleiche Effekt kann auch über eine ori-Instruktion erreicht werden, der Befehl move hilft aber deutlich der Lesbarkeit des eigenen Codes.

```
move $t0, 100            //speichert die Konstante 100 im Register $t0

=

ori $t0, $zero, 100      // das OR einer Konstante mit 0 ist wieder die Konstante selber. Diese
                         // wird im Zielregister $t0 abgelegt.
```

Die direkten Befehle unserer Assemblerprogrammiersprache können als das sogenannte „ Instruktion Set“ des Rechners interpretiert werden. Das Instruktion Set eines Rechners sind alle Operationsbefehle an den Rechner, die direkt in eine 32-Bit breite Maschinencode-Instruktion übersetzt werden können. Diese Maschinencode-Instruktionen werden dann innerhalb eines einzigen Clock-Zyklus im Rechner ausgeführt. Es gibt Rechnerstrukturen mit einigen hundert Instruktionen (RISC: Reduced Instruction Set) oder einigen tausend verschiedene Instruktionen (CISC: Complex Instruction Set). Für unseren Rechner haben wir das Instruction Set in Abb. 2.8 nochmals zusammengefasst. In den Beispielen werden nur direkte Befehle, also Befehle des Instruction Set verwendet.

Instruction Set

Operation	Assembler-Befehl	Wirkung	Format
Addition	add $c, $a, $b	$c = $a + $b	R
Subtraktion	sub $c, $a, $b	$c = $a – $b	R
Logisches AND	and $c, $a, $b	$c = $a & $b	R
Logisches OR	or $c,$a, $b	$c = $a | $b	R
Set on Less than	slt $c, $a, $b	if $a < $b then $c = 1 else $c = 0	R
Addition Im.	addi $c, $a, K	$c = $a + K	I
Subtraktion Im.	subi $c, $a, K	$c = $a – K	I
Logisches AND Im.	andi $c, $a, K	$c = $a & K	I
Logisches OR Im.	ori $c,$a, K	$c = $a | K	I
Set on Less than Im.	slti $c, $a, K	if $a < K then $c = 1 else $c = 0	I
Load Word	lw $c, K($a)	$c = Mem[K + $a]	I
Store Word	sw $c, K($a)	Mem[K + $a] = $c	I
Branch on equal	beq $a, $b, K	if $a == $b then PC = K	I
Branch on not equal	bne $a, $b, K	if $a <> $b then PC = K	I
Jump	j K	PC = K	J
Jump and Link	jal K	$ra = PC + 4, PC = K	J
Jump Register	jr $ra	PC = $ra	R

Abb. 2.8: Tabelle mit dem Instruction Set für den Rechner. Diese Operationen können durch eine entsprechende Maschinencode-Instruktion abgearbeitet werden.

Aufgabe 2.4:

- Der indirekte Befehl „branch on lower than“ (bl), verzweigt auf eine konstante Sprung-Adresse, falls ein Wert in Register a kleiner als ein Wert in einem Register b ist. Schreiben Sie die Anweisung mit direkten Befehlen.
- Der indirekte Befehl „branch on greater/equal than“ (bge), verzweigt auf eine konstante Sprung-Adresse, falls ein Wert in Register a grösser oder gleich wie ein Wert in einem Register b ist. Schreiben Sie die Anweisung mit direkten Befehlen.
- Der indirekte Befehl „Increase“ (inc), nimmt als Parameter eine Registeradresse und erhöht den Wert des Registers konstant um 1.
- Der indirekte Befehl „Decrease“ (dec), nimmt als Parameter eine Registeradresse und vermindert den Wert des Registers konstant um 1.

2.2 Programm-Anweisungen

In diesem Abschnitt wollen wir den Umgang mit den Assembler-Befehlen etwas auffrischen und die gängigsten Programm-Anweisungen in Assembler übersetzten. Wir betrachten dabei zuerst eine einfache Addition zweier Konstanten, danach die Subtraktion zweier Werte aus dem Data Memory. Darauf folgt ein Beispiel wie in Arrays mit dem Offset zugegriffen werden kann. Danach implementieren wir die Statements „If-then-else“, und „while-Loop“. Wir werden im folgenden häufig die Register $t0-$t7 verwenden, dies sind die Register, die für Operanden frei benutzt werden können. Die weiteren Benutzungskonventionen der Register werden etwas später eingeführt.

Addition zweier Konstanten:

```
c = 45 + 39;                    // Addition zweier Konstanten in Pseudocode

=

ori $t0, $zero, 45              // Lade Konstante 45 in Register $t0
ori $t1, $zero, 39              // Lade Konstante 39 in Register $t1
add $t0, $t0, $t1               // Addiere Register $t0 und $t1. Das Resultat wird in Register $t0
                                // gespeichert.
```

Subtraktion zweier Werte aus dem Data Memory:

Um die Werte aus dem Data Memory laden zu können, brauchen wir deren Adresse. Wir nehmen an, dass die Adressen für den Wert A und B in den Registern $a0 und $a1 enthalten sind. Das Ergebnis wird in die Adresse C aus Register $a2 zurückgeschrieben. Für die Berechnung von Adressen im Data Memory werden Pointer (auf deutsch: „Zeiger“) verwendet. Der Stack- und der FramePointer (sp und fp) werden für lokale Datenstrukturen benutzt, der GlobalPointer (gp) wird für die globalen Datenstrukturen verwendet. Die zwei Zeiger fp und sp werden wir bei den Prozedurwechsel noch etwas genauer kennen lernen.

```
Mem[C+0] =Mem[A+0] –  Mem[B+0]    // Subtraktion zweier Werte aus dem Daten Memory in
                                  // Pseudocode. Der Offset ist jeweils 0, da es sich nicht um
                                  // Arrayzugriffe handelt.
=

lw $t0, 0($a0)                    // Wert A mit Adresse $a0 wird aus dem Memory gelesen und in
                                  // das Register $t0 gespeichert
lw $t1, 0($a1)                    // Wert B mit Adresse $a1 wird aus dem Memory gelesen und in
                                  // das Register $t1 gespeichert
sub $t0, $t0, $t1                 // Der Wert aus Register $t1 wird vom Wert aus Register $t0
                                  // subtrahiert. Das Resultat wird im Register $t0 gespeichert.
sw $t0, 0($a2)                    // Der Wert aus Register $t0 wird im Memory unter der Adresse $a2
                                  // gespeichert.
```

Aufgabe 2.5: Schreiben Sie ein Assembler-Programm, das aus der Data Memory-Adresse A in $a0 den Wert ausliest, und zu diesem Wert den konstanten Wert 100 addiert. Das Resultat wird im Data Memory unter der Adresse B in $a1 gespeichert.

Addition zweier Werte aus einem Array:

Nehmen wir an, die Variable A ist ein Array der Grösse 3 im Data Memory. Es können darin also zum Beispiel 3 verschiedene Integerwerte à jeweils 4 Byte abgespeichert werden. Die 3 Felder des Arrays können wie gewohnt mit einem von 0 aus startenden Index adressiert werden. Wir wollen nun die Felder A[0] und A[1] addieren und im Feld A[2] speichern. Die Memory Adresse des ersten Feldes wird wieder in Register $a0 angenommen.

```
Mem[A+2] = Mem[A+0] + Mem[A+1]     // Addition zweier Arraywerte aus dem Daten Memory in
                                   // Pseudocode
```

=

```
lw $t0, 0($a0)          // Lade den Wert des ersten Feldes des Arrays A aus dem Memory in
                        // das Register $t0
lw $t1, 4($a0)          // Lade den Wert des zweiten Feldes des Arrays A aus dem Memory
                        // in das Register $t1. Um auf den nächst höheren Index des Arrays
                        // zuzugreifen, muss die Größe eines Arrayfeldes bei der
                        // Adressberechnung mit eingerechnet werden. Daher muss die Basis-
                        // Adresse in $a0 um die Konstante 4 erhöht werden.
add $t0, $t0, $t1       // Addiere die Werte aus Register $t0 und $t1. Das Resultat wird in
                        // Register $t0 abgespeichert.
sw $t0, 8($a0)          // Speichert den Wert aus Register $t0 im dritten Feld des Arrays A.
```

Aufgabe 2.6: Im Register $a0 ist die Adresse des ersten Feldes eines Arrays A gespeichert. Schreiben Sie ein Programm, dass im 5. Feld des Arrays den Wert 24 addiert.

"if then else" Statement:

Wir nehmen an, dass sich die Variablen i und j in den Registern $t0 und $t1 befinden. Die Variablen f, g und h befinden sich in den Registern $t5, $t6 und $t7. Wir wollen nun die folgende Verzweigung implementieren:

```
if (i == j) then            // Eine „If-then-else“ Verzweigung in Pseudocode. Je nach
        f = g + h;          // Auswertung der Bedingung i == j wird eine Addition oder eine
else                        // Subtraktion auf die Variablen in den Registern ausgeführt.
        f = g – h;
end
```

=

```
bne $t0, $t1, ELSE      // Springe zur Zeile mit dem Label “ELSE” falls i ungleich j ist
add $t5, $t6, $t7       // i ist gleich j, daher werden die Variablenwerte addiert
j END                   // Springe zur Zeile mit dem Label „END“
ELSE: sub $t5, $t6, $t7 // Die Zeile ist mit dem Label „ELSE“ markiert. i ist ungleich j, daher
                        // werden die Variablenwerte subtrahiert
END:                    // Die Zeile markiert durch das Label „END“ das Ende des „If-then-
                        // else“ Statements. Der Resultatwert der Variable f ist in Register $t5.
```

Aufgabe 2.7: Schreiben Sie ein Assembler-Programm für das folgende Switch-Case Statement. Je nach Wert einer „switch“-Variable k werden die entsprechenden Befehle im dazugehörigen „case“-Block abgearbeitet. Existiert kein entsprechender case-Block, so werden die Befehle im case: default abgearbeitet. Die Werte der Variablen k, x, y und z sind in den Register $a0-$a3 gegeben.

```
switch k: {
        case k = 0: { d = x + y};
        case k = 1: { d = x – z};
        case k = 2: { d = y + z};
        case default: { d = y – z};
};
```

“while” loop Statement:

Beim „while-loop“ Statement wird solange eine Befehlssequenz wiederholt, bis eine bestimmte Bedingung erfüllt worden ist. Im Beispiel wird ein Integer-Array A nach einem bestimmten Wert w durchsucht. Falls der Wert gefunden wird, springt das Programm zur Zeile mit dem Label „OK“. Wird der Wert nach 25 Versuchen (die Grösse des Arrays) nicht gefunden, so springt das Programm zur Zeile „FAILED“. Die Basis-Adresse für das Array A wird in $a0 angenommen, der zu vergleichende Wert w wird in $a1 erwartet.

```
i = 0;                                  // Pseudocode eines „while-loop“ Beispieles. Es wird in einem
while (Mem[A+i] <> w && i < 25)         // Array der Grösse 25 nach einem Wert w gesucht.
        i = i + 1;
end;
if (Mem[A+i] == w) then
        OK
else
        FAILED
end;
```

=

```
ori $t0, $zero, $zero       // Die Variable i wird Register $t0 zugewiesen und mit 0 initiiert
LOOP: add $t1, $t0, $a0     // Die Variable i wird als Offset mit der Basis-Adresse von A addiert
                            // und im Register $t1 gespeichert.
lw $t2, 0($t1)              // Das Arrayfeld A[i] wird aus dem Memory ins Register $t2 geladen
beq $t2, $a1, OK            // Falls A[i] gleich w ist, wird auf die Zeile mit dem Label OK
                            // gesprungen
slti $t3, $t0, 100          // Falls i kleiner als 100 ist (25 * Feldgrösse 4 Byte) so wird in Register
                            // $t3 eine 1 ausgegeben
beq $t3, $zero, FAILED      // Falls $t3 gleich 0 ist, dann ist i grössergleich 100 und es muss zur
                            // Zeile mit Label FAILED gesprungen werden
addi $t0, $t0, 4            // Wurde nicht aus der Schleife gesprungen, so wird der Index i in
                            // Register $t0 um ein Feld, also 4 Bytes erhöht.
j LOOP                      // Rücksprung um die Schleife nochmals auszuführen
OK: .....                   // In dieser Zeile fährt das Programm fort, falls im Array ein Wert gleich
                            // w gefunden werden konnte
FAILED: ...                 // In dieser Zeile fährt das Programm fort, falls im Array kein Wert
                            // gleich w gefunden werden konnte
```

Aufgabe 2.8: Schreiben Sie ein Assembler-Programm für das folgende Do-while Statement, das sämtliche Werte eines Arrays A auf 0 setzt. Mit „do“ wird der Beginn einer Befehlssequenz markiert. Das „while“ wertet eine Bedingung aus, aufgrund der entschieden wird, ob die Befehlssequenz wiederholt werden soll. Die Befehle werden also mindestens einmal abgearbeitet. Die Basis-Adresse für A wird in $a0 erwartet, die Grösse des Arrays „A.size“ ist in $a1 angegeben.

```
i = 0;
do {
        A[i] = 0;
        i = i + 1;
}
while ( i < A.size)
```

2.3 Prozeduraufrufe

Der gesamte Programmcode wird in sogenannte „Prozeduren“ unterteilt. Eine Prozedur umfasst normalerweise den Code zu einer bestimmten, häufig gebrauchten Funktion die meistens einen Resultatwert zurückgibt. Wenn ein Programm gestartet wird, wird Immer zuerst eine Prozedur mit Namen „main“ gestartet. Von dieser aus können weitere Prozeduren aufgerufen werden, die dann abgearbeitet werden bis ein Resultat zurückgegeben wird. Damit sich die Prozeduren nicht gegenseitig Register, lokale Variablen oder ganze Datenstrukturen im Data Memory überschreiben, wird im Daten Memory ein sogenannter „Stack“ (der deutsche Begriff ist „Datenstapel“) angelegt.

Auf dem Stack werden die sogenannten „StackFrames“ der Prozeduren angelegt. Ein StackFrame ist der von einer Prozedur benötigte Speicherplatz um den gesamten Stand der Prozedur zu speichern während eine andere aufgerufene Prozedur abgearbeitet wird. Der oberste StackFrame ist immer der Speicherbereich für die gerade laufende Prozedur. Es wird also immer nur am obersten StackFrame manipuliert. Der Stack wächst etwas unintuitiv von hohen Speicheradressen zu niedrigeren. Soll der StackFrame z.B. vergrößert werden, muss dies mit einer Subtraktion des StackPointers geschehen. Was der StackPointer genau ist und wie bei einem Prozeduraufruf ein neues StackFrame erstellt wird zeigen die folgenden Schritte:

1. Wenn eine Prozedur A aufgerufen wird erhält sie eine sogenannte „FramePointer-Adresse“ (fp) sowie eine „Rücksprung-Adresse“ (ra). Ab der FramePointer-Adresse darf die Prozedur Speicherplatz z.B. für lokale Variablen belegen. Wieviel Speicher die Prozedur für sich belegt hat, wird mit dem StackPointer (sp) markiert. Der StackPointer zeigt dabei immer auf die erste Adresse im freien Speicherbereich. Eine Prozedur kann nun weitere Variable oder Datenstrukturern speichern bzw. löschen, indem sie den StackPointer um den benötigten Platz erhöht bzw. erniedrigt (vom sp subtrahiert bzw. addiert). Die StackPointer-Adresse ist nie größer als die FramePointer-Adresse. Die Variablen können nun im neu angelegten Platz gespeichert werden. Der Bereich zwischen FramePointer und StackPointer nennt sich „StackFrame“ der Prozedur.

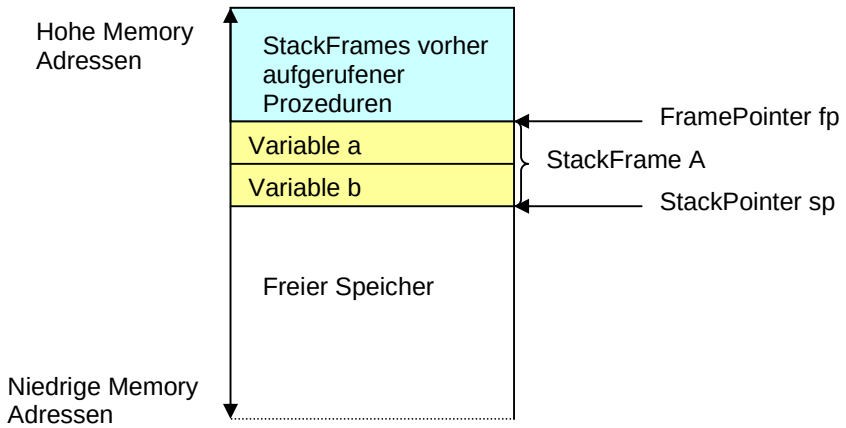


Abb. 2.9: Ein Stack in dem im aktuellen StackFrame die aktuelle Prozedur zwei Variablen a und b gespeichert hat

2. Kommt es nun zu einem weiteren Prozeduraufruf B aus der laufenden Prozedur A, dann muss sich die aktuelle Prozedur A den gesamten momentanen Stand speichern. Dies sind mindestens die Register $s0-$s7 für lokale Variablen, die Rücksprungadresse zur vorhergegangenen Prozedur und der FramePointer. Eventuell werden aber noch ganze Datenstrukturen im Stack abgelegt. Eine Begründung, warum die Register $s0-$s7 immer gespeichert werden, finden sie in Kapitel 6.
3.

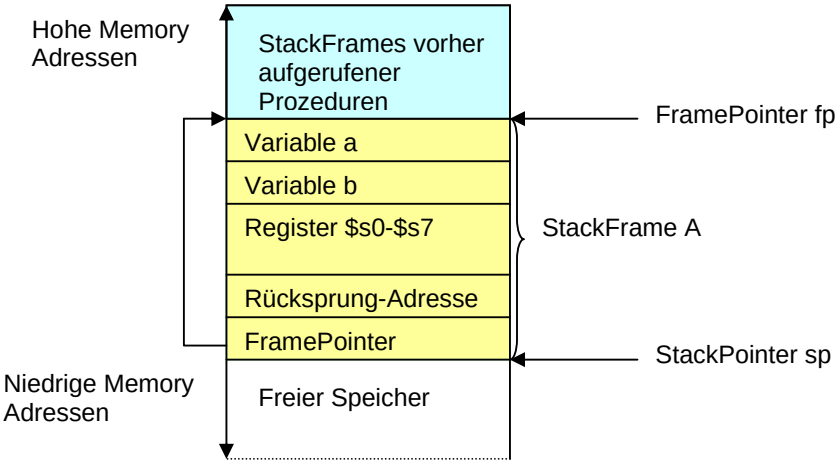


Abb. 2.10: Bei einem Prozeduraufruf werden die Register $s0-$s7 sowie die Rücksprung-Adresse auf eine Instruktion und der FramePointer gespeichert. Der Platz für die Register $s0-$s7 wird auch angelegt, wenn die Registerwerte nicht gespeichert werden müssten

4. Nun müssen die Argumente von der Prozedur A an die neue Prozedur B übergeben werden. Dazu werden die Argumente in die Register $a0-$a3 geschrieben. Reichen die Register nicht aus um alle Argumente zu übergeben, dann muss dies über den Stack geschehen. Auch wenn ein Resultat von der Prozedur B erwartet wird, das nicht mit den Resultat-Registern $v0 und $v1 zurückgegeben werden kann, muss freier Speicher im StackFrame vorgemerkt werden. Wir gehen auf dieses interessante Thema hier aber nicht weiter ein.

5. Die Prozedur A nimmt nun den StackPointer und macht diesen zum neuen FramePointer für die Prozedur B (Siehe Punkt 1). Die Rücksprung-Adresse zu A für die Prozedur B wird mit dem Befehl „Jump and Link“ übergeben.

6. Mit dem Befehl „Jump and Link“ (jal) wird nun also auf die gewünschte Instruktionsadresse der aufgerufenen Prozedur B gesprungen. Gleichzeitig wird die aktuelle Instruktionsadresse (PC + 4) im Register $ra gespeichert, damit die aufgerufene Prozedur B wissen kann, zu welcher Instruktion zurückgesprungen wird, wenn die Prozedur beendet ist und die Prozedur A weiter abgearbeitet werden kann. Dies entspricht bildlich Abb. 2.11. Durch Erhöhen des StackPointers (mittels einer Subtraktion) kann die Prozedur B nun neuen Speicherplatz reservieren und nutzen. Abb. 2.12.

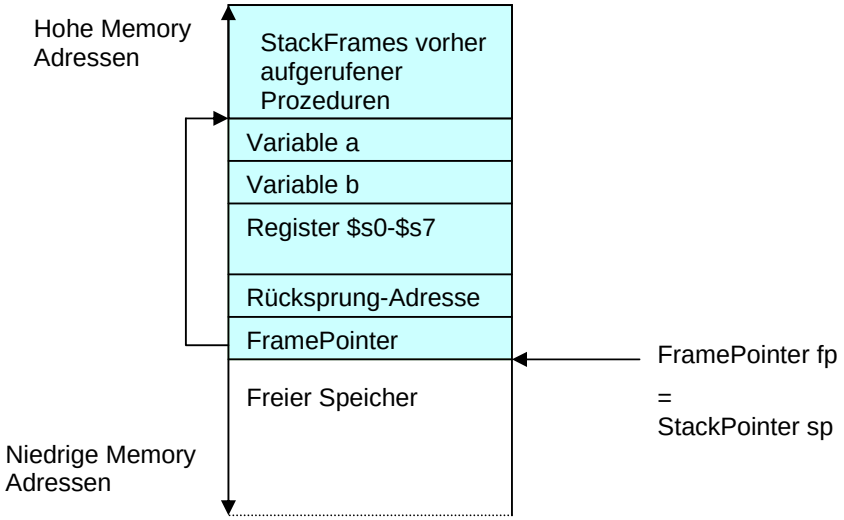


Abb. 2.11: Um das aktuelle StackFrame aufzubauen bzw. abzubauen wird der FramePointer = StackPointer gesetzt.

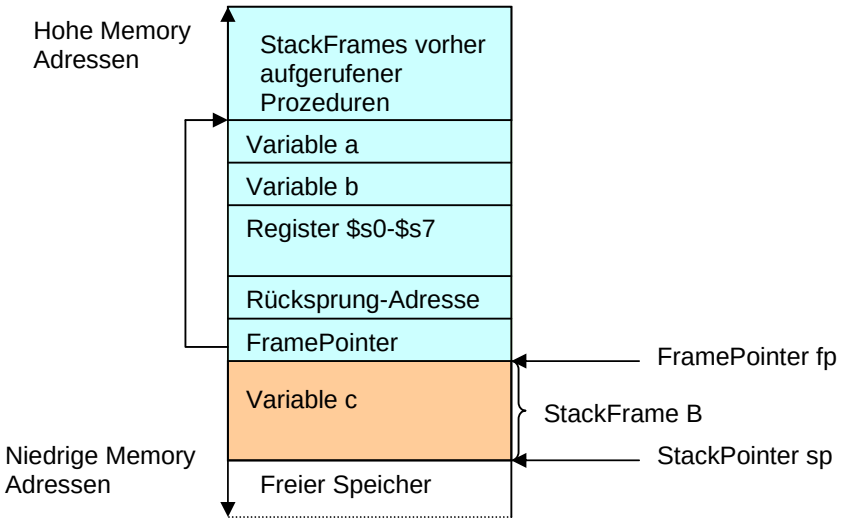


Abb. 2.12: Die aufgerufene Prozedur nimmt als FramePointer den StackPointer der vorherigen Prozedur. Nun kann mit dem StackPointer die Prozedur Speicherplatz reservieren. In der Abbildung hat die Prozedur B eine Variable c gespeichert.

7. Hat die Prozedur B ein Resultat berechnet, so wird dieses in den Registern $v0 und $v1 abgelegt. Danach wird das eigene StackFrame abgebaut, indem einfach der StackPointer = FramePointer gesetzt wird. Dies entspricht wieder Abbildung 2.11.

8. Durch den Befehl „Jump Register“ kann von der Prozedur B wieder zu der alten Prozedur A zurückgesprungen werden, da im Register $ra die Rücksprung-Adresse gespeichert wurde.

9. Die alte Prozedur A kann nun den eigenen gespeicherten FramePointer und die eigene Rücksprung-Adresse wieder in die entsprechenden Register zurückladen, sowie die nicht mehr benötigten zwischengespeicherten Werte vom eigenen StackFrame abbauen, wie in Abb. 2.13. Durch den StackPointer kann der nicht mehr benötigte Speicherplatz wieder freigegeben werden. Dann sieht der Stack wieder aus wie in Abb. 2.9. Danach kann die Prozedur weiter abgearbeitet werden.

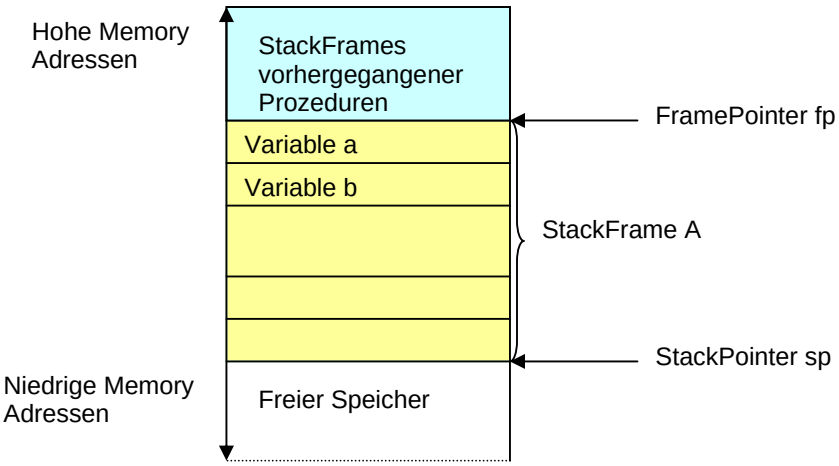


Abb. 2.13: Prozedur A stellt den gespeicherten FramePointer und die Rücksprungadresse wieder her. Auch die gespeicherten Registerwerte $s0-$s7 können wieder in die Register zurückgeladen werden. Der StackPointer hat in der Abbildung den nicht mehr benötigten Speicher noch nicht freigegeben.

Der Ablauf eines Prozeduraufrufes/Rücksprungs nennt sich **Calling Convention**. Diese kann sich von Maschine zu Maschine etwas unterscheiden.

Aufgabe 2.9:

a) Welche Bedingung muss beim Aufrufen oder Beenden einer neuen Prozedur auf dem Stack erfüllt sein?
b) Welche Werte müssen im Stack-Frame einer Prozedur zwingend gespeichert werden?
c) Für welche Werte werden bei unserer Calling Convention Platz auf dem Stack angelegt, obwohl dies nicht unbedingt notwendig wäre.

Wir wollen nun an einem Beispiel zeigen, wie die Prozeduraufrufe mit dem Stack zusammenspielen. Dazu programmieren wir zuerst eine Multiplikationsfunktion.

```
int Multiplikation (int a, int b)        // Die Prozedur berechnet die Multiplikation zweier Werte a
        c = 0 ;                          // und b mit den Operationen Addition und Subtraktion
        while (b >= 1) {
                c = c + a ;
                b = b – 1 ;
        }
        return c;
}

=

MUL: or $t0, $zero, $zero                // Register $t0 wird der Variable c zugewiesen und mit 0
                                         // initiiert
LOOP: slti $t1, $a1, 1                   // prüft ob b grössergleich 1 ist, das Resultat  wird in $t1
                                         // gespeichert. Der Wert  der Variable b wird in Register $a1
                                         // erwartet
bne $t1, $zero, END                      // falls $t1 ungleich null ist, wird der „while-Loop“ beendet
add $t0, $t0, $a0                        // Die Variable c wird um den Wert a erhöht. Die Variable a
                                         // wird im Register $a0 erwartet
subi $a1, $a1, 1                         // Die Variable b wird um 1 vermindert
j LOOP                                   // Die Schleife wird wiederholt
END: or $v0, $t0, $zero                  // Die Variable c ist das Resultat und wird in das Rückgabe-
                                         // Register $v0 geschrieben.
jr $ra                                   // Rücksprung zur vorhergegangenen Prozedur
```

Und nun ein kleines Programm, das unsere Multiplikationsprozedur aufruft und ein Integer-Resultat zurückgibt. Nennen wir es „main“

```
int main () {                          // Ein Programm führt eine Multiplikationen aus. Der Wert der
        u = 5;                         // Variablen u und v muss beim  Prozeduraufruf gespeichert
        v = 3;                         // werden, da die Variablen noch benötigt werden
        w = Multiplikation (u, v);
        x = u + v;
        return w – x;
}

=

MAIN: ori $t0, $zero, 5                // Variable u in Register $t0 mit 5 initialisiert
ori $t1, $zero, 3                      // Variable v in Register $t1 mit 3 initialisiert
subi $sp, $sp, 8                       // StackFrame wird für Variable u und v um 8 Byte vergrößert
sw $t0, 0($fp)                         // Variable u wird zu unterst im StackFrame gespeichert
sw $t1, -4($fp)                        // Variable v wird als nächstes im StackFrame gespeichert

Stack sieht  nun aus wie in Abb. 2.9

subi $sp, $sp, 40                      // Vergrössert den Stack für die Register $s0-$s7,
                                       // die Rücksprung-Adresse und den FramePointer um
                                       // 8 * 4 Byte ($s0-$s7) + 4 Byte($ra) + 4 Byte ($fp) = 40 Byte
                                       // Die Register $s0 -$s7 müssen nicht gespeichert werden, der
                                       // Speicherplatz wird trotzdem reserviert
sw $ra, -40($fp)                       // Rücksprung-Adresse wird im StackFrame gespeichert
sw $fp, -44($fp)                       // FramePointer Adresse wird gespeichert

Stack sieht nun aus wie in Abb. 2.10
or $a0, $t0, $zero                     // Verschiebe 1. Argument von $t0 nach $a0
or $a1, $t1, $zero                     // Verschiebe 2. Argument von $t1 nach $a1
or $fp, $sp, $zero                     // FramePointer = StackPointer

Stack sieht nun aus wie in Abb. 2.11

jal MUL                                // Prozeduraufruf. In $ra wird die Rücksprung-Adresse aus
                                       // der Prozedur MUL gespeichert.

                                       // Resultat von MUL in wird Register $v0 gespeichert und
                                       // hierher zurückgesprungen

lw $fp, 4($sp)                         // FramePointer wiederherstellen
lw $ra, 8($sp)                         // Rücksprung-Adresse wiederherstellen

Stack sieht nun aus wie in Abb. 2.13

addi $sp, $sp, 40                      // StackFrame abbauen bis auf Variable u und v

Stack sieht wieder aus wie in Abb. 2.9

lw $t0, 0(fp)                          // Variable u aus Stack in Register $t0
lw $t1, -4(fp)                         // Variable v aus Stack in Register $t1

add $t2, $t0, $t1                      // Addiere Variable u und v. Resultat in Register $t2
sub $v0, $v0, $t2                      // Subtrahiere das  Additionsresultat vom Resultat aus MUL
                                       // Resultat wird im Rückgaberegister $v0 gespeichert.

or $sp, $fp, $zero                     // StackFrame ganz abbauen: $sp = $fp
jr $ra                                 // Prozedur beendet. Rücksprung
```

Aufgabe 2.10: Schreiben Sie eine Assembler-Prozedur, die die Fakultät einer Zahl berechnet. Zeichnen Sie den Stack bei der Abarbeitung Ihres Programms. Die Fakultät von z.B. 5 ist 5 * 4 * 3 * 2 * 1 und wird mit 5! angeschrieben. Sie können für die Multiplikationen die vorgestellte Prozedur benutzen. Das Programm kann wie folgt geschrieben werden.

```
int Fakultät (int a){
        b = 1;
        while (a > 1){
                b = Multiplikation (b, a);
                a = a –1 ;
        }
        return c ;
}
```

Aufgabe 2.11: Schreiben Sie eine Assembler-Prozedur, die die Fakultät einer Zahl berechnet. Zeichnen Sie den Stack bei der Abarbeitung Ihres Programms. Schreiben Sie diesmal aber die Prozedur rekursiv, d.h. eine Prozedur die sich selber aufruft. Das Programm kann wie folgt geschrieben werden.

```
int Fakultät_R (int x){
        if (x == 1) return 1;
        else {
                y = Fakultät_R (x-1);
                return Multiplikation (x, y);
        };
} ;
```

Zusammenfassung

Wir haben in diesem Kapitel zuerst Assemblerbefehle vorgestellt und diese gemäss ihren benötigten Parameter in drei verschiedene Klassen „R-Format“, „I-Format“ und „J-Format“ unterteilt. Wir haben auch gesehen, dass bestimmte Befehle direkt in eine Maschinencode-Instruktion für den Rechner übersetzt werden können, während andere Befehle in ein oder zwei Befehle umgewandelt werden müssen, damit sie dann in Maschinencode übersetzt werden können. Die direkt übersetzbaren Befehle gehören zum Instruction Set des Rechners.

Im zweiten Abschnitt sollte das Programmieren in Assembler etwas aufgefrischt werden. Wir haben gesehen wie Operationen mit Konstanten, Variablen aus dem Memory oder aus Registern ausgeführt werden können. Wir haben als Beispiel für Offset-Konstanten eine Operation auf Array-Felder vorgestellt, und danach die Programm-Statements „if-then-else“ sowie einen „while-Loop“ implementiert.

Im letzten Abschnitt haben wir den Stack vorgestellt, der uns hilft den Speicher so zu organisieren, dass sich verschiedene Prozeduren nicht gegenseitig Daten überschreiben. Die einzelnen Schritte auf dem Stack und den Registern bei einem Prozeduraufruf haben gezeigt was alles nötig ist, wenn aus einer Prozedur eine andere aufgerufen wird. Der genaue Ablauf der einzelnen Schritte bei einem Prozeduraufruf wird „Calling Convention“ genannt. Als Beispiel haben wir eine Prozedur „Multiplikation“ programmiert, und diese aus einem Programm „main“ aufgerufen.

Lernziele

Haben Sie die folgenden Lernziele des Kapitels erfüllt? Falls Sie sich nicht ganz sicher fühlen, schlagen Sie die Punkte nochmals nach. Fühlen Sie sich beim Umgang mit dem Stoff sicher? Dann können Sie vorwärts zum Kapiteltest blättern.

- Sie wissen nach welchen Kriterien die Assembler-Befehle in Klassen eingeteilt werden
- Sie wissen, was das Instruction Set eines Rechners ist
- Sie können mit Assembler einfache Programme erzeugen
- Sie wissen, für was ein Stack nötig ist, und wie der Stack und Register bei Prozeduraufrufen manipuliert werden
- Sie wissen, was unter dem Begriff Calling Convention verstanden wird.

Kapiteltest 2:

Aufgabe 1:

Der Befehl „branch on lower/equal than" (ble) verzweigt auf eine konstante Sprung-Adresse, falls ein Wert in Register a kleiner oder gleich wie ein Wert in einem Register b ist. Schreiben Sie die Anweisung mit direkten Befehlen aus dem Instruction Set.

Aufgabe 2:

Schreiben Sie eine Prozedur in Assembler mit dem Namen „Invert". Diese Prozedur nimmt als Parameter einen Zeiger auf eine Adresse im Daten Memory. Die Prozedur liest den 32-Bit Wert an dieser Memory-Adresse und speichert das Zweierkomplement des Wertes wieder an derselben Memory-Adresse. Die Prozedur-Deklaration könnte wie folgt aussehen:

*void Invert (*int adresse);*

Aufgabe 3:

Schreiben Sie eine Prozedur mit dem Namen „Exp". Diese Prozedur nimmt zwei Parameter a und b entgegen und gibt als Resultat den Wert a hoch b zurück. Nehmen Sie dafür an, dass eine Multiplikationsprozedur mit der folgenden Deklaration bereits vorhanden ist:

int Multiplikation (int a, int b);

Aufgabe 4 :

Zeichen Sie einen Stack mit 3 Stackframes. Was ist in den 3 Stackframes mindestens gespeichert?

Kapitel 3: Maschinencode

Einleitung

In diesem Kapitel wollen wir die Brücke zwischen Hard- und Software schlagen. Es wird dabei der sogenannte „Maschinencode“ für die Befehle des Instruction Set erzeugt. Wir beleuchten dabei für jede Format-Klasse die Opcodes der Befehle und zeigen die Sprungadress-Berechnungen für die einzelnen Befehle auf. Im Kapitel ist jeder Format-Klasse ein Abschnitt zugewiesen. Ein vierter Abschnitt beinhaltet ein komplettes Beispiel eines Programms in Maschinencode. Zuerst wird aber wieder eine kleine Einführung gegeben.

1 Im ersten Abschnitt werden die Befehle der „J-Format“ Klasse betrachtet. Diese Klasse enthält nur eine Konstante als Argument für Programm-Sprünge. Wir werden sehen wie die Sprungadresse aus dieser Konstante berechnet werden kann.

2 Der zweite Abschnitt ist der Klasse „I-Format“ zugeteilt. Die Argumente der Klasse sind zwei Register sowie eine Konstante. Die Adressberechnung für einen LoadWord oder StoreWord Befehl wird hier nicht all zu grosse Kopfschmerzen bereiten, jedoch die Adressberechnung bei „branch on equal“ und „branch on not equal“ Befehlen geben Stoff fürs Gehirn.

3 In Abschnitt 3 wird dann noch die Klasse „R-Format“ vorgestellt. Hier werden 3 Registeradressen als Argumente erwartet. In diesem Abschnitt werden wir eine Technik vorstellen, wie mit weiterem freien Platz in der Instruktion der Opcode entlastet werden kann.

4 Am Schluss des Kapitels werden wir die Programme „Multiplikation“ und „main“ in Maschinencode übersetzten und in einem Instruction Memory ablegen. Dadurch soll gezeigt werden, wie die Instruktionen nun physikalisch im Memory aussehen, und wie die Adressberechnung genau funktioniert. Wir zeigen also was die Arbeit eines Compilers und eines sogenannten „Linkers“ ist.

Einführung

Es wurde bereits erwähnt, das eine Abfolge von Assembler-Befehlen für einen Rechner noch unverständlich sind. Im Kapitel 2 wurde das Instruction Set zu unserem Rechner zusammengestellt, also alle Befehle die direkt in eine Maschinencode-Instruktion übersetzt werden können. Was ist nun so eine Maschinencode Instruktion? Im Kapitel 1 haben wir uns ein Instruction Memory zugelegt, das in jedem Takt-Zyklus ein 32-bit Signal von einer Adresse ausliest. Dieses 32-bit Signal ist eine solche Instruktionen. In der Instruktion müssen einerseits die Argumente zum Befehl enthalten sein, sowie eine Anweisung an den Rechner, welche Kontrollsignale alle gesetzt werden müssen. Der Teil der Instruktion, der für die Kontrollsignalsteuerung zuständig ist, wird „**Opcode**“ genannt. Der Opcode identifiziert also einen Befehl für den Rechner. Wir folgen einer Konvention zu den Instruktionformaten wie sie in MIPS Rechnerstrukturen verwendet werden, und belegen für den Opcode 6 Bits der Instruktion. Mit 6 Bits können 2 hoch 6 = 64 verschiedene Anweisungen übertragen werden. Wie können aber einige hundert Instruktionen von einem Rechner unterschieden werden? Im Abschnitt 3 werden wir ein Beispiel sehen, wo der Opcode nur eine Klasse von Instruktionen bestimmt, und die benötigte Instruktion aus einem nicht für Operanden benutzten Feld bestimmt werden kann. Zuerst wollen wir uns aber den Befehlen der Klasse „J-Format“ zuwenden.

3.1 J-Format Befehle

Die Klasse der J-Format Befehle benötigt als Argument nur eine möglichst grosse Konstante. Die Konstante wird als Sprungadresse für die nächste Instruktion interpretiert. Die Instruktion dieser Klasse beinhaltet also nur 6 Bits für den Opcode sowie die restlichen 26 Bits für die Adresse.

Bit 31 -26	Bit 25 - 0
Opcode	Sprung-Adresse

Abb. 3.1: Instruktionsformat der „J-Format“ Klasse. Die 6 obersten Bits beinhalten den Opcode des Befehls, die Bits 25 - 0 sind die Konstante für die Sprungadresse.

Mit den 26 Bits der Sprungadresse kann nun aber nicht auf alle 2 hoch 32 Adressen im Instruction Memory gesprungen werden. Wir müssen uns deshalb etwas einschränken. Erstmals können aber der Sprungadresse zwei Nullen angehängt werden, da eine Instruktionsadresse durch 4 teilbar sein muss. Damit haben wir also die Sprungadresse von 26 auf 28 Bits erweitert und mit 4 multipliziert. Es verbleiben noch 4 Bits. Da die Sprungziele meistens nicht weit weg von der aktuellen Instruktionsadresse liegen, werden die obersten 4 Bits von der aktuellen Instruktion (PC+4) verwendet. Dadurch erhalten wir einen erreichbaren Bereich für die Sprungziele relativ zum aktuellen PC, die Adresse innerhalb dieses Bereiches ist aber mit der Konstante absolut angegeben. Der Compiler ist dafür verantwortlich, das die Sprungziele im erreichbaren Bereich im Instruction Memory zu liegen kommen.

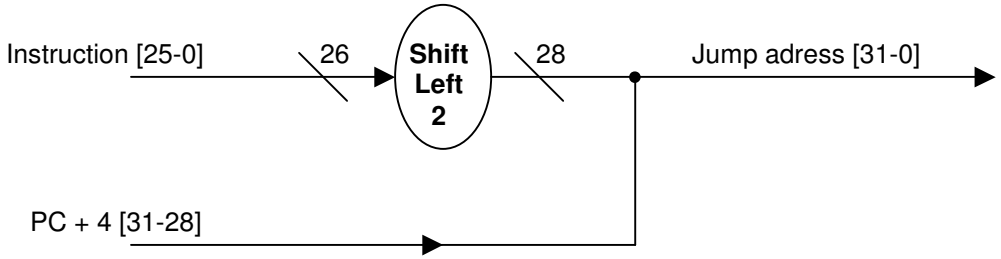


Abb. 3.2: Berechnung der Sprungadresse bei der Klasse „J-Format“. Die Adresse aus der Instruktion wird um 2 Leitungen erweitert und mit 4 multipliziert, die beiden Most Significant Bits werden also nicht abgeschnitten. Das gleiche Ergebnis wäre erhältlich, wenn zwei Leitungen mit Wert 0 als Least Significant Bits hinzugefügt werden. Die 4 höchsten Bitpositionen [31-28] werden vom PC + 4 übernommen. Damit ist ein relativer Bereich zum PC von 2 hoch 28 Instruktionsadressen adressierbar.

Nehmen wir an, wir befinden uns bei der Instruktionsadresse 10110011010011001111111111010100. Wir möchten nun auf eine Instruktion mit der Adresse 10110000111110001010101011001100 springen. Welche Konstante muss nun in der Instruktion angegeben werden? Die 4 Most Significant Bits kommen direkt aus der Instruktionsadresse: [1011]00001111100010101010011001100. Die 2 Least Significant Bits sind immer 0 und kommen aus der Adresserweiterung und Multiplikation mit 4: [1011]00001111100010101010110011[00]. Das was noch übrig bleibt ist unsere 26-bit Adress-Konstante: 00001111100010101010110011. Die Adressberechnung für Sprünge wird nicht vom Programmierer gemacht, da dieser normalerweise nicht weiss, wo das Programm im Instruktionsspeicher zu liegen kommt. Wenn ein Programm in den Instruktionsspeicher geladen wird, wird ein Programm, der „Linker“ aufgerufen. Dieser kann die Label im Maschinencode dann durch wirkliche Adressen ersetzten.

Wir brauchen jetzt noch einen Opcode um die „Jump“ (j) und die „Jump and Link“ (jal) Instruktion von den übrigen Instruktionen zu unterscheiden. Wir haben im Opcode 6 Bits zur Verfügung und können nun bestimmte Opcodesignale reservieren: Für den Befehl „Jump“ reservieren wir den Opcode 6: 000110 und für „Jump and Link“ verwenden wir den Opcode 7: 000111.

Opcodes der Klasse J-Format

Operation	Assembler Befehl	Opcode binär	Opcode dezimal
Jump	j	000110	6
Jump and Link	jal	000111	7

Abb. 3. 3: Tabelle der Opcodes zur Klasse J-Format. Die Operationen der Klasse wurden mit dem dazugehörigen Assemblerbefehl angegeben, sowie ist der Opcode zu den Befehlen in Binär und Dezimalwerten wiedergegeben.

Aufgabe 3.1: Berechnen Sie die benötigte 26 Bit Konstante für einen J-Format Sprungbefehl, wenn von der aktuellen Instruktionsadresse (PC + 4) mit Wert 2’416’000’000 auf die Adressen a) 2’415’910’000 b) 2’415’990’000 c) 2’600’000’000 d) 2’690’000’000 gesprungen werden soll.

3.2 I-Format Befehle

Die Befehle der I-Format Klasse haben als Argumente zwei Register-Adressen und eine Konstante. Der Opcode benötigt von der 32-Bit breiten Instruktion 6 Bits, und die zwei Registeradressen benötigen jeweils 5 Bits. Es verbleiben für die Konstante also noch 16 Bits.

Bit 31 -26	Bit 25 - 21	Bit 20 - 16	Bit 15 - 0
Opcode	Register A	Register B	Konstante

Abb. 3.4: Instruktionsformat der „I-Format" Klasse. Die 6 obersten Bits beinhalten den Opcode des Befehls, die Bits 25-21 sind die Adresse des Resultat-Registers. In den Bits 20 – 16 ist die Adresse des Argumentregisters gegeben. Die Konstante ist in den Bits 15 –0 zu finden. Bei Branch-Instruktionen werden die beiden Register verglichen und die Konstante als relative Sprungadresse zu aktuellen PC genutzt.

Betrachten wir zuerst die Operationen „addi", „subi", „ori", „andi" und „slti". Die Konstante wird als binärer Wert der ALU zugeführt, sie muss dafür noch mit einer „Sign-Extension" auf 32 Bits erweitert werden.

Bei der Load-Word und Store-Word Instruktion wird die Konstante als Offset zu einer 32-Bit Memory Adresse addiert, die aus einem Register kommt. Hier ist die eigentliche Adresse in einem Register gespeichert, und damit kann also problemlos auf alle 2 hoch 32 möglichen Memory-Adressen zugegriffen werden. Mit der Konstante wird also nur das Programmieren von Array-Feld-Zugriffen vereinfacht, wie es in Kapitel 2 gezeigt wurde.

Schwieriger wird es bei den „branch on equal" und „branch on not equal" Befehlen. Hier wird die Konstante wieder als Sprungziel-Adresse benötigt. Zuerst kann die Adresse wieder mit 4 multipliziert werden, da die Instruktionen nur in durch 4 teilbaren Memory-Adressen beginnen. Es fehlen nun aber noch 14 Adress-Bits. Auch hier wird wieder der aktuelle Programm-Counter (PC + 4) verwendet, um einen relativen Bereich zum PC adressierbar zu machen. Es werden aber nicht einfach wie bei der Jump-Adressierung die fehlenden Bits vorne angehängt, sondern das ganze Programm-Counter Signal wird mit der Adress-Konstante addiert. Dadurch entsteht wieder ein relativ erreichbarer Bereich zum Programm-Counter, die Konstante in der Instruktion ist nun aber keine absolute Adresse in diesen Bereich sondern auch eine relative.

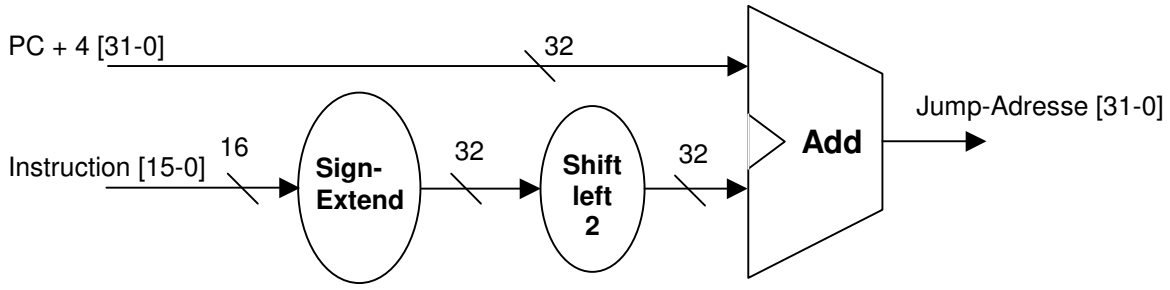


Abb. 3.5: Adressberechnung bei „branch on equal" und „branch on not equal" Befehlen. Die Adress-Konstante wird von 16 auf 32 Bits erweitert und dann mit einem Shift Left 2 mit 4 multipliziert. Das Ergebnis ist die relative Adresse zum aktuellen Programm-Counter. Diese zwei Werte werden mit einem Addierer addiert.

Dazu wieder ein kleines Beispiel: nehmen wir an, der PC + 4 Wert ist bei der momentan ausgeführten Branch Instruktion: 10110011010011001111111111010100. Wir wollen bei erfüllter Verzweigungsbedingung auf die Instruktionsadresse 10110011010010111101011010001000 springen. Wir wollen nun rückwärts die nötige Konstante berechnen.

Als zweiter Operand am Addierer ist also die Sprung-Adresse – (PC + 4) anzulegen:
10110011010010111101011010001000 - 10110011010011001111111111010100.

Das Zweierkomplement von PC + 4 ist: 01001100101100110000000000101100.

Nun die Addition mit der Sprung-Adresse: 10110011010011111101011010011000 + 01001100101100110000000000101100 = 11111111111111101101011010110100.

Die Division durch 4 verschiebt die Stellen des Binärwertes um 2 Stellen nach rechts. Dabei wird links der Wert des bisherigen Most Significant Bit eingefügt: 11111111111111111011010110101101.

Die ersten 16 Bits aus dem Sign-Extend werden abgeschnitten und wir erhalten die benötigte Konstante: 1011010110101101.

Die benötigten relativen Sprungadressen werden auch wieder vom Linker in dem Maschinencode vervollständigt. Da der Sprungbereich mit 16 Bits eher klein ist und man wirklich einen weiteren Sprung benötigt, kann man sich in Assembler mit einem kleinen Trick behelfen.

```
bne $t0, $t1, TARGET        // Mit einer bedingten Sprunginstruktion soll auf das Label
                            // TARGET gesprungen werden, die Adress-Konstante reicht
                            // für die Weite des Sprungs aber nicht aus
=

bne $t0, $t1, FARJUMP       // bedingter Sprung auf eine nahe gelegene unbedingte
FARJUMP: j TARGET           // Sprunginstruktion. Das Format der unbedingten
                            // Sprunginstruktionen erlaubt weiter entfernt liegende
                            //Sprungziele zu erreichen
```

Wir brauchen nun noch einen Opcode für die Instruktionen der I-Format Klasse. Diese sind in der folgenden Tabelle aufgeführt.

Operation	Befehl	Register A	Register B	Konstante	Opcode Binär	Opcode Dezimal
Addition Im.	addi	Resultat der ALU Op	Operand ALU	Operand ALU	001000	8
Subtraktion Im.	subi	Resultat der ALU Op	Operand ALU	Operand ALU	001001	9
Logisch OR Im.	ori	Resultat der ALU Op	Operand ALU	Operand ALU	001010	10
Logisch AND Im.	andi	Resultat der ALU Op	Operand ALU	Operand ALU	001011	11
set on less than Im.	slti	Resultat der ALU Op = 1 oder 0	Operand ALU	Operand ALU	001100	12
Load Word	lw	Wert aus Memory	Basis-Adresse	Offset	001101	13
Store Word	sw	Wert in Memory	Basis-Adresse	Offset	001110	14
Branch on equal	beq	Vergleichs-Operand ALU	Vergleichs-Operand ALU	Sprung-Adresse	010000	16
Branch on not equal	bne	Vergleichs-Operand ALU	Vergleichs-Operand ALU	Sprung-Adresse	010001	17

Abb. 3.6: Tabelle der Operationen der I-Format Klasse. In der Tabelle ist zu sehen, welche Bedeutung die Argumente einer Instruktion haben. Der Opcode zu den einzelnen Operationen ist in den Spalten rechts in binärer und dezimaler Darstellung angegeben

Aufgabe 3.2: Berechnen Sie die benötigte 16 Bit Konstante für einen I-Format Sprungbefehl „branch on equal“ oder „branch on not equal“. Die aktuelle Instruktionsadresse (PC + 4) ist 2’416’000’000. Von dieser aus soll auf die Adressen a) 2’415’910’000 b) 2’415’990’000 c) 2’600’000’000 d) 2’690’000’000 gesprungen werden.

Aufgabe 3.3: Zeichnen Sie sich schematisch ein Instruction Memory und weisen Sie einer Adresse den aktuellen PC zu. Zeichnen Sie nun einen Bereich ein, der mit einer J-Format Instruktion adressiert werden kann. Nun bestimmen Sie eine nicht weit entfernte Adresse zum aktuellen PC, die 4 obersten Bits sind dabei dieselben. Zeichen Sie wiederum den mit J-Format adressierbaren Bereich ein. Nun zeichnen Sie noch ein Instruction Memory und wiederholen die Anleitung, diesmal aber für I-Format „branch on equal“ und „branch on not equal“-Adressierungen. Wie unterscheiden sich die adressierbaren Bereiche?

3.3 R-Format Befehle

Kommen wir zum verbleibenden Format, den Befehlen des R-Formates. Wir haben bei einem R-Format Befehl 3 Register-Adressen als Argumente. Von den 32 Instruktionsbits werden 6 Bits wieder für den Opcode verwendet, und 3 * 5 Bits werden für die Registeradressen benötigt. Es verbleiben also noch 11 Bits der Instruktion ungenutzt. Wir können nun nur einen Opcode für die gesamte Klasse von R-Format Instruktionen benutzen, und die bestimmte Operation über den noch freien Bereich der Instruktion identifizieren lassen. Dadurch können jetzt weit mehr als die 2 hoch 6 = 64 Operationen mit einer Instruktion identifiziert werden. Wir unterteilen die 11 noch freien Bits in 2 Felder: das Feld „funct“ ist 6 Bit breit und das Feld „shamt“ belegt die restlichen 5 Bits.

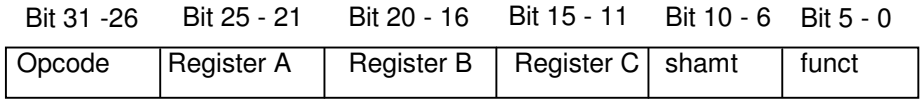


Abb. 3.7: Instruktionsformat der „R-Format“ Klasse. Die 6 obersten Bits beinhalten den Opcode des Befehls. Der Opcode ist für alle Befehle des R-Formats gleich. Die Bits 25-21 sind die Adresse des Resultat-Registers. In den Bits 20 – 16 ist die Adresse des ersten Argumentregisters gegeben. Die Adresse des zweiten Argumentregisters ist in den Bits 15 –11 zu finden. Das Feld „shamt“ von Bit 10 –6 wird bei uns nicht genutzt und ist deshalb immer 0. Durch das Feld „funct“ in den Bits 5-0 bestimmt die auszuführende Operation aus der R-Format Klasse.

Das Feld „shamt“ wird in der MIPS Konvention für Shifts (Verschiebungen) von Binärwerten um eine variable Anzahl Stellen benutzt. Da unsere ALU diese Operation nicht kennt, ist dieses Feld bei uns immer 0. Mit Feld „funct“ können wir die R-Format Operationen identifizieren. Die folgende Tabelle zeigt, die Belegung der Felder „Opcode“ und „funct“. Die Registeradresse A wird immer als Resultatregister benutzt, die anderen zwei Registeradressen beinhalten die Werte der ALU Operanden.

Operation	Befehl	Opcode binär	Opcode dezimal	funct binär	funct dezimal
Addition	add	100000	32	000100	4
Subtraktion	sub	100000	32	000101	5
Logisch AND	and	100000	32	000000	0
Logisch OR	or	100000	32	000010	2
Set on Less than	slt	100000	32	000111	7
Jump Register	jr	100000	32	001000	8

Abb. 3.8: Tabelle mit den Operationen der „R-Format“ Klasse. Der Opcode ist für alle Operationen derselbe, d.h. der Opcode identifiziert hier nur eine Klasse von Operationen. Mit dem Feld „funct“ in den Instruktionen der R-Format-Klasse wird die auszuführende ALU Operation bestimmt. Die Werte im Feld „funct“ entsprechen den Kontrollsignalen an die ALU. Siehe hierfür Abb. 1.21.

Damit haben wir alle Befehle unseres Instruction Sets mit einem Opcode belegt und das Format der einzelnen Instruktionsklassen festgelegt. Die Tabelle in Abb. 3.9 fasst alle Opcodes nochmals zusammen.

Operation	Befehl	Format	Opcode	funct
Addition	add	R	32	4
Subtraktion	sub	R	32	5
Logisch AND	and	R	32	0
Logisch OR	or	R	32	2
Set on less than	slt	R	32	7
Jump Register	jr	R	32	8
Addition Im.	addi	I	8	-
Subtraktion Im.	subi	I	9	-
Logisch AND Im.	andi	I	10	-
Logisch OR Im.	ori	I	11	-
Set on less than Im.	slti	I	12	-
Load Word	lw	I	13	-
Store Word	sw	I	14	-
Branch on equal	beq	I	16	-
Branch on not equal	bne	I	17	-
Jump	j	J	6	-
Jump and Link	jal	J	7	-

Abb. 3.9: Zusammenfassung der Opcodes. Bei R-Format Instruktionen ist der Opcode immer 32. Die Instruktionsoperation wird durch das Feld „funct“ dieser Instruktionsklasse eindeutig bestimmt.

Aufgabe 3.4: Wieviele Befehle können mit den angegebenen Instruktionsformaten maximal unterschieden werden?

3.4 Beispiel eines Maschinencodes

Wir haben im Kapitel 2 ein Programm “Multiplikation” und ein Programm “main” in Assembler implementiert, und wollen nun die Assembler-Befehle in Maschinencode Instruktionen umwandeln. Den Maschinencode werden wir dann ins Instruction Memory laden.

Hier nochmals die Programme:

```
int Multiplikation (int a, int b)          =          MUL: or $t0, $zero, $zero
        c = 0 ;                                       LOOP: slti $t1, $a1, 1
        while (b >= 1) {                              bne $t1, $zero, END
                c = c + a ;                           add $t0, $t0, $a0
                b = b – 1 ;                           subi $a1, $a1, 1
        }                                             j LOOP
        return c;                                     END: or $v0, $t0, $zero
}                                                     jr $ra
```

```
int main () {                          =        MAIN: ori $t0, $zero, 5
        u = 5;                                  ori $t1, $zero, 3
        v = 3;                                  subi $sp, $sp, 8
        w = Multiplikation (u, v);              sw $t0, 0($fp)
        x = u + v;                              sw $t1, -4($fp)
        return w – x;                           subi $sp, $sp, 40
}                                               sw $ra, -40($fp)
                                                sw $fp, -44($fp)
                                                or $a0, $t0, $zero
                                                or $a1, $t1, $zero
                                                or $fp, $sp, $zero
                                                jal MUL
                                                lw $fp, 4($sp)
                                                lw $ra, 8($sp)
                                                addi $sp, $sp, 40
                                                lw $t0, 0(fp)
                                                lw $t1, -4(fp)
                                                add $t2, $t0, $t1
                                                sub $v0, $v0, $t2
                                                or $sp, $fp, $zero
                                                jr $ra
```

Wir werden nun die Instruktionen für jedes Programm in Maschinencode übersetzten. Die Label und Speicheradressierungen interessieren uns erst im nächsten Schritt.

Betrachten wir die erste Zeile aus dem Programm „Multiplikation“: MUL: or $t0, $zero, $zero.

Den Zeilenmarker belassen wir auf weiteres noch:	=> MUL:	
Die Instruktion ist von Typ R-Format, mit Opcode:	32	100000
Das Zielregister ist $t0 ($r8, siehe Abb. 1.24):	8	01000
Die beiden Operanden-Register sind beide $zero ($r0):	0	00000
		00000
Das shamt-Feld von R-Format Instruktionen bleibt immer	0	00000
Das Feld funct der R-Format Instruktion ist	2	000010

Der Maschinencode lautet folglich: =>MUL: 100000 01000 00000 00000 00000 000010

Auf die gleiche Weise bestimmen wir den Maschinencode für die zweite Instruktion: LOOP: slti $t1, $a1, 1

Zeilenmarker:	=>LOOP:	
Opcode für I-Format slti:	12	001100
Register $t1 ($r9)	9	01001
Register $a1 ($r5)	5	00101
Konstante 1	1	0000000000000001

Der Maschinencode lautet folglich: =>LOOP: 001100 01001 00101 0000000000000001

Das folgende Bild zeigt die Programme im Instruction Memory. Es wurde angenommen, dass das Programm „main“ ab der Adresse 0, und das Programm „Multiplikation“ ab der Adresse 2048 im Instruction Memory gespeichert wird.

Instruktion:	Adresse:	Label:	Maschinencode:
MUL: or $t0, $zero, $zero	2048	MUL	100000 01000 00000 00000 00000 000010
LOOP: slti $t1, $a1, 1	2052	LOOP	001100 01001 00101 0000000000000001
bne $t1, $zero, END	2056		010001 01001 00000 END
add $t0, $t0, $a0	2060		100000 01000 01000 00100 00000 000100
subi $a1, $a1, 1	2064		001001 00101 00101 0000000000000001
j LOOP	2068		000110 LOOP
END: or $v0, $t0, $zero	2072	END	100000 00010 01000 00000 00000 000010
jr $ra	2076		100000 11111 00000 00000 00000 001000

Instruktion:	Adresse:	Label:	Maschinencode:
MAIN: ori $t0, $zero, 5	0000	MAIN	001011 01000 00000 0000000000000101
ori $t1, $zero, 3	0004		001011 01001 00000 0000000000000011
subi $sp, $sp, 8	0008		001001 11101 11101 0000000000001000
sw $t0, 0($fp)	0012		001110 01000 11110 0000000000000000
sw $t1, -4($fp)	0016		001110 01001 11110 1111111111111100
subi $sp, $sp, 40	0020		001001 11101 11101 0000000000101000
sw $ra, -40($fp)	0024		001110 11111 11110 1111111111011000
sw $fp, -44($fp)	0028		001110 11110 11110 1111111111010100
or $a0, $t0, $zero	0032		100000 00100 01000 00000 00000 000010
or $a1, $t1, $zero	0036		100000 00101 01001 00000 00000 000010
or $fp, $sp, $zero	0040		100000 11110 11101 00000 00000 000010
jal MUL	0044		000111 MUL
lw $fp, 4($sp)	0048		001101 11110 11101 0000000000000100
lw $ra, 8($sp)	0052		001101 11111 11101 0000000000001000
addi $sp, $sp, 40	0056		001000 11101 11101 0000000000101000
lw $t0, 0(fp)	0060		001101 01000 11110 0000000000000000
lw $t1, -4(fp)	0064		001101 01001 11110 1111111111111100
add $t2, $t0, $t1	0068		100000 01010 01000 01001 00000 000100
sub $v0, $v0, $t2	0072		100000 00010 00010 01010 00000 000101
or $sp, $fp, $zero	0076		100000 11101 11110 00000 00000 000010
jr $ra	0080		100000 11111 00000 00000 00000 001000

Abb. 3.10: Ein Programm wurde in Maschinencode übersetzt und an bestimmte Adressen im Instruction Memory geladen. Die Sprungadressen bei den Label im Maschinencode wurden noch nicht aufgelöst.

Wir haben soweit die Arbeit des Compilers übernommen und die Assembler-Befehle in Maschinencode übersetzt. Der Compiler löst die Sprungadressen bei den Labels noch nicht auf, sondern hinterlegt Verweise zu den verschiedenen Labels. Wenn der Maschinencode in das Instruction Memory geladen wird, so wie es in Abb. 3.10 der Fall ist, kann der Linker damit beginnen die Sprungadressierungen bei den Labels im Maschinencode zu berechnen und im Maschinencode einzufügen.

Wir haben drei Labels im Maschinencode für die Prozedur Multiplikation, bei denen wir nun die Sprungadressierungen berechnen müssen: Bei Adresse 2056 das Label END, bei Adresse 2068 das Label LOOP und bei Adresse 044 das Label MUL. Beginnen wir mit dem Label END:

* Bei Adresse 2056, dem Label END, soll auf die Adresse 2072 gesprungen werden. Das Label END steht in einem „Branch on not equal“ Befehl, also einem I-Format Befehl. Die Sprungadresse ist also in einer 16-Bit Konstante anzugeben.

Jump-Adresse:	2072	00000000000000000000100000011000
PC + 4:	2060	00000000000000000000100000001100
(Jump-Adresse) – (PC+4)	12	00000000000000000000000000001100
/4	3	00000000000000000000000000000011
-Sign-Extension		0000000000000011

⇨ Die 16-Bit Konstante beim Label END in Adresse 2056 ist also der Wert 3: 0000000000000011.

* Bei Adresse 2068, dem Label LOOP, soll auf die Adresse 2052 gesprungen werden. Das Label LOOP steht in einem Jump-Befehl, also ein J-Format Befehl. Die Sprungadresse wird also in einer 26-Bit Konstante gespeichert.

Jump-Adresse:	2052	00000000000000000000100000000100
- 4 Most Significant Bits (PC+4)		00000000000000000100000000100
- 2 Least Significant Bits		00000000000000001000000001

⇨ Die 26-Bit Konstante beim Label LOOP in Adresse 2068 ist also der Wert 513: 00000000000000000100000001.

* Zum Schluss haben wir noch im Programm „main“ bei Adresse 044 das Label MUL. Hier soll also auf die Adresse 2048 gesprungen werden. Der Befehl an der Adresse 044 ist „Jump and Link“, also wieder ein J-Format Befehl mit einer 26-Bit breiten Sprungadresse:

Jump-Adresse	2048	00000000000000000000100000000000
- 4 Most Significant Bits(PC)		00000000000000000100000000000
- 2 Least Significant Bits		00000000000000001000000000

⇨ Die 26-Bit Konstante beim Label MUL in Adresse 044 hat den Wert 512: 00000000000000001000000000.

Wird nun dieses Programm gestartet, so muss dem Programm ein gültiger FramePointer sowie eine sinnvolle Rücksprung-Adresse zugewiesen werden. Danach ist alles bereit für die Abarbeitung der Instruktionen.

Aufgabe 3.5: Erstellen Sie den kompletten Maschinencode für ihre zwei Programme zur Fakultätsberechnung aus Kapitel 2. Schreiben Sie dazu auch ein Programm „main“ in Assembler, das zuerst die nicht-rekursive Version mit dem Wert 5 aufruft, danach die rekursive Version ihrer Programme mit dem vorher berechneten Resultat aufruft. Übersetzten Sie auch dieses in Maschinencode. Geben Sie die Adressen an, an denen ihre Programme im Instruction Memory gespeichert werden und füllen Sie die benötigten Sprung-Adressen ein.

Zusammenfassung:

In diesem Kapitel wurden Assembler-Befehle in Maschinencode-Instruktionen übersetzt. Dazu haben wir nacheinander die Instruktionsformate der Befehlsklassen angeschaut. Wir haben in der Instruktion ein spezielles Feld angegeben um für den Rechner die einzelnen Befehle unterscheidbar zu machen: den 6-Bit breiten Opcode.

Dann haben wir bei den J-Format-Befehlen die Sprungadressierung studiert und gemerkt, dass nicht auf jede mögliche Adresse im Instruction Memory gesprungen werden kann, sondern nur in einem zum PC absoluten Bereich. Die Sprungadresse ist in einem Instruktionsargument der Breite 26 Bit angegeben.

Bei den I-Format-Befehlen haben wir die Befehle „Branch on equal“ und „Branch on not equal“ genauer betrachtet. Auch diese beinhalten eine Sprungadressierung auf das Instruction Memory. Hier wurde die Sprungadressierung mit einer 16-Bit Konstante angegeben, und lässt dadurch einen relativen Bereich zum aktuellen PC als Sprungziele zu. Die Adressierung mit LoadWord und StoreWord zielt aufs Data Memory ab, und ist mit einem Register als Basis-Adresse einfach und uneingeschränkt zu bewältigen.

Bei der Klasse R-Format wurde aufgezeigt, wie der Opcode nur eine Klasse von Befehlen identifiziert und die eigentliche Operationsunterscheidung über ein weiteres Feld „funct“ erfolgt. Diese Technik hilft uns weit mehr Operationen für den Rechner unterscheidbar zu machen, als es das Feld des Opcodes alleine erlauben würde.

Zum Abschluss des Kapitels haben wir an den zwei Programmen „Multiplikation“ und „main“ aus Kapitel 2 gezeigt, wie die Assembler-Befehle nun in Maschinencode übersetzt werden können. Die Sprungadressierungen wurden aber noch nicht berechnet. Erst wenn der Instruktionscode im Instruction Memory geladen ist kann das Linker-Programm die nötigen Sprungkonstanten einsetzen. Am Schluss erhielten wir den vollständigen Maschinencode für ein Programm im Instruction Memory, der nun bereit wäre abgearbeitet zu werden.

Lernziele

Haben Sie die folgenden Lernziele des Kapitels erfüllt? Falls Sie sich nicht ganz sicher fühlen, schlagen Sie die Punkte nochmals nach. Fühlen Sie sich beim Umgang mit dem Stoff sicher? Dann gehen Sie weiter zum Kapiteltest.

- Sie wissen, für was der Opcode einer Instruktion benötigt wird
- Sie wissen, wie man einen Assembler-Befehl aus dem Instruction Set in einen Maschinencode übersetzen kann
- Sie wissen, wie die Sprung-Adresse je nach Befehlsklasse aus einer Instruktionskonstante und dem momentanen Program Counter berechnet werden kann
- Sie wissen, warum Sprungverzweigungen nicht von Anfang an im Maschinencode aufgelöst werden, sondern warum diese mit Labels angezeigt werden, um erst vor der Ausführung des Codes aufgelöst zu werden.

Kapiteltest 3

Aufgabe 1:

Berechnen Sie die benötigte 26 Bit Konstante für einen J-Format Sprungbefehl, wenn von der aktuellen Instruktionsadresse (PC + 4) mit dem binären Wert 10101111000011110000111100001100 auf die binären Adressen

a) 10100000111100001111000011110000
b) 10101111111111111111111111111100

gesprungen werden soll.

Aufgabe 2:

Berechnen Sie die benötigte 16 Bit Konstante für einen I-Format Sprungbefehl „branch on equal“ oder „branch on not equal“. Die aktuelle Instruktionsadresse (PC + 4) hat den binären Wert 10110000000000000000000000000000

a) 10101111110000001111000011110000
b) 10110000001111111100000000000000

gesprungen werden soll.

Aufgabe 3:

Welchem Zweck dient das Feld „funct“ in einer R-Format Instruktion?

Aufgabe 4:

Übersetzen Sie folgende Assembler-Befehle in den entsprechenden Maschinencode

a) addi $t2, $t1, 222
b) sub $sp, $sp, $s0
c) lw $a0, 8($t0)
d) jal LABEL
e) bne $s2, $s4, LABEL

Kapitel 4: Datenpfade

Einleitung

Da wir nun die Formate der Maschinencode-Instruktionen kennen, können wir nun beginnen einen Rechner zusammenzubauen, der uns diese Instruktionen ausführen kann. Alle benötigten Komponenten wurden in Kapitel 1 vorgestellt, diese müssen jetzt nur noch richtig zusammengesetzt werden. In diesem Kapitel wird das Augenmerk nur auf die Datenleitungen gelegt, das heisst, die Steuerung der Kontrollsignale wird noch nicht betrachtet. Die Steuerung des Rechners, also das richtige setzen der Kontrollsignale über den Opcode wird durch eine sogenannte „Control Logic“ ausgeführt, die wir in Kapitel 5 zusammenstellen werden. Dieses Kapitel ist in 6 Abschnitte eingeteilt:

1 Im ersten Abschnitt wird ein Rechner gebaut, der die ALU-Instruktionen mit R-Format unterstützt.

2 Im zweite Abschnitt erweitern wir den Rechner, so dass die ALU-Instruktionen mit dem I-Format ausgeführt werden können.

3 In Abschnitt 3 fügen wir dem Rechner das Data Memory bei. Wir werden dabei die nötigen Erweiterungen des Rechners vorstellen, um die Befehle LoadWord und StoreWord auszuführen.

4 Der 4. Abschnitt beschäftigt sich mit der Umsetzung von „Branch on equal“ und „Branch on not equal „Instruktionen.

5 Die beiden Instruktionen des J-Formates werden im 5. Abschnitt umgesetzt. Zuerst wird die Jump-Instruktion unterstützt, danach können wir mit einer kleinen Erweiterung die Instruktion „Jump and Link“ ausführen lassen.

6 Der letzte Abschnitt komplettiert die Datenleitungen des Rechners: es wird die noch fehlende Instruktion „Jump Register“ des R-Formates in die Rechnerstruktur eingebaut.

4.1 ALU-Instruktionen mit R-Format

Wir wollen zuerst einen Rechner bauen, der in jedem Clock-Zyklus eine Instruktionen aus dem Instruction Memory liest. Es sollen die direkten Assembler-Befehle add, sub, and, or und slt unterstützt werden. Die Instruktionen sind vom Typ R-Format, haben also die Zielregister-Adresse für das ALU-Resultat in den Bits 25-21. Die erste Argument-Registeradresse liegt in den Bits 20-16 die zweite Argument-Registeradresse ist in den Bits 15-11 zu finden. Wir haben bereits im Kapitel 1 zwei Schaltungen vorgestellt: Die erste liest uns die Instruktionen aus dem Instruction Memory, die zweite kombinierte den Register-Block mit der ALU. Wir müssen nun nur noch diese zwei Schaltungen zusammensetzten und die richtigen Instruktionsbits an die richtigen Eingänge des Register-Blocks leiten. Die Schaltung wird in Abb. 4.1 gezeigt.

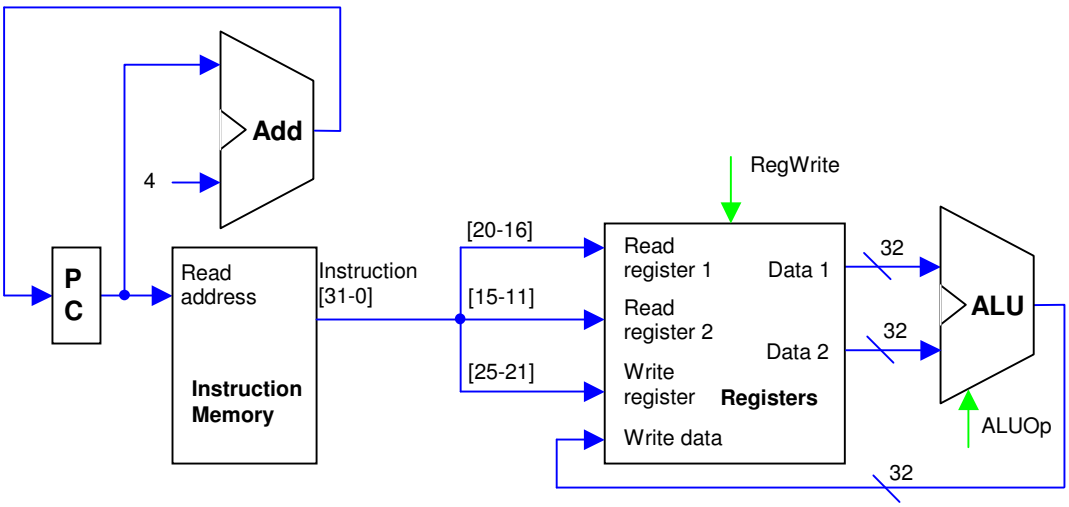


Abb. 4.1: Schaltung für einen Rechner der die R-Format Instruktionen add, sub, and, or, und slt unterstützt.

4.2 ALU-Instruktionen mit I-Format

Wir erweitern nun die Schaltung um die Möglichkeit, die ALU-Instruktionen addi, subi, andi, ori und slti auszuführen. Beim I-Format dieser Maschinencode-Instruktionen wird die Zielregisteradresse in den Bits 25-21 gespeichert. Diese Bits sind bereits am „Write register"-Eingang des Register-Blocks angeschlossen. Auch die erste Operand-Registeradresse in den Bits 20-16 der Instruktion ist beim Eingang „Read register 1" richtig angeschlossen. Als zweiten Operanden für die ALU-Instruktion soll nun aber die Konstante aus den Bits 15-0 verwendet werden. Ein Multiplexer am ALU-Eingang für den zweiten Operanden erlaubt uns die Wahl, ob wir den Operanden aus einem Register oder die Konstante wollen. Das Kontrollsignal für den Multiplexer nennen wir „ALUSrc". Da die Konstante 16 Bits breit ist, der Operand für die ALU aber 32 Bits breit sein muss, muss die Konstante mit einer Sign-Extension auf die 32 Bits erweitert werden. Die nötigen Erweiterungen sind in Abb. 4.2 hervorgehoben.

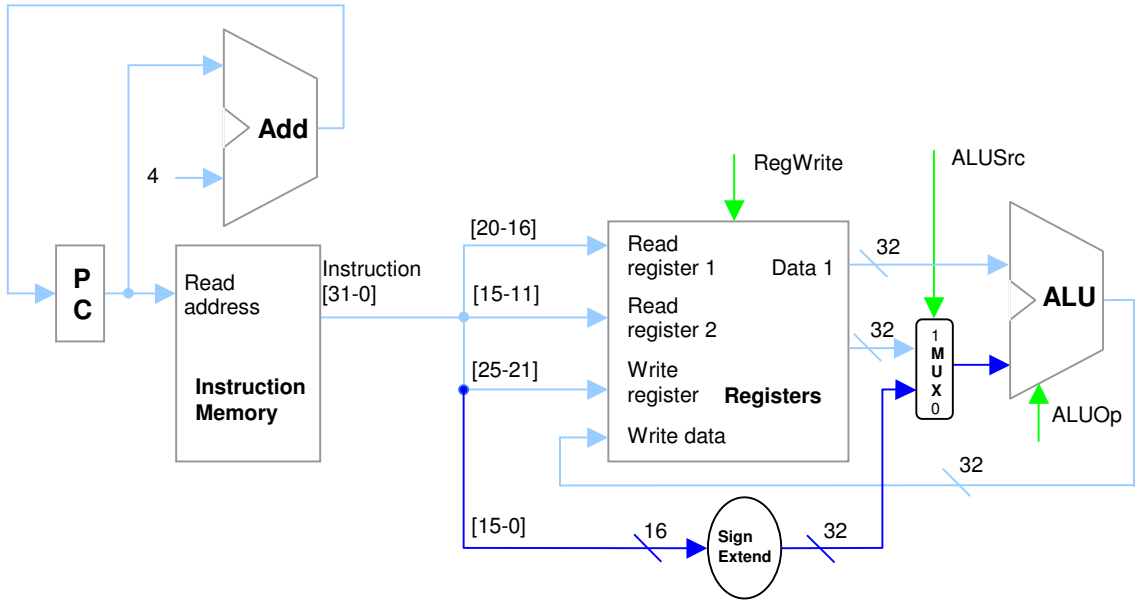


Abb. 4.2: Schaltung für einen Rechner der die R-Format Instruktionen add, sub, and, or, und slt sowie die I-Format Instruktionen addi, subi, andi, ori, slti unterstützt. Die nötigen Erweiterungen sind hervorgehoben.

4.3 LoadWord und StoreWord

Für eine LoadWord oder StoreWord Instruktion erweitern wir die Schaltung mit einem Data Memory. Die Memory-Adresse wird mit einer I-Format Addition berechnet, d.h. die ALU addiert den Registerwert der Register-Adresse in Bit 20-16 mit der Offset-Konstante aus Bit 15-0. Das Resultat wird in den Data Memory-Eingang „address“ eingespiesen.

Bei einer LoadWord Operation soll nun der Memorywert an der berechneten Adresse vom Memory-Ausgang „data“ zum Register mit der Adresse in Bit 25-21 zur Speicherung geleitet werden. Um den Wert aus dem Memory auszulesen, wird das Kontrollsignal „MemRead“ notwendig sein. Durch einen Multiplexer mit Kontrollsignal, RegWriteSrc“ muss dazu gesteuert werden können, ob das ALU-Resultat gespeichert werden soll oder der aus dem Memory ausgelesene Wert. In Abb. 4.3 ist die Erweiterung für die LoadWord Operation dargestellt.

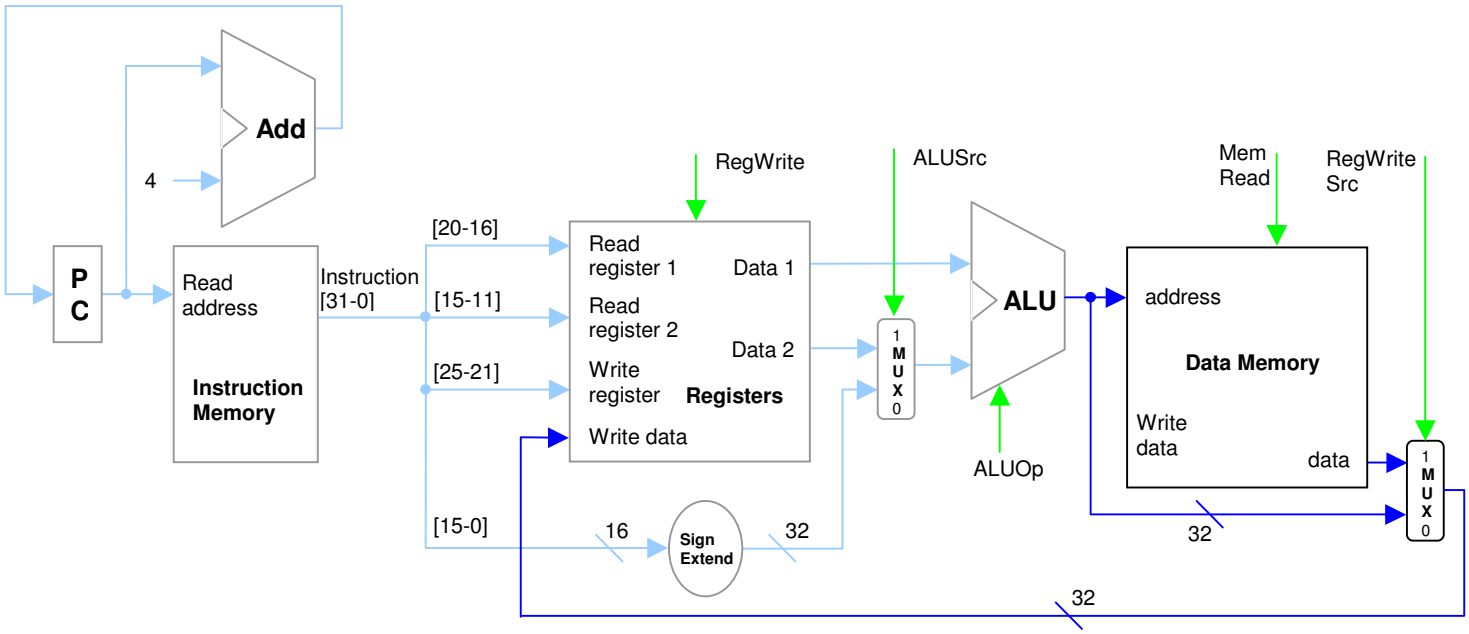


Abb. 4.3: Schaltung mit der hinzugefügten Komponente Data Memory. Die Erweiterungen für die I-Format Instruktion LoadWord sind hervorgehoben.

Aufgabe 4.1: Bevor Sie weiterlesen, überlegen Sie sich, welche Erweiterungen nötig werden, um eine StoreWord Instruktion auszuführen. Sie können dafür das bis hierhin erstellte Schaltbild beliebig erweitern. Die Auflösung zu dieser Aufgabe ist im folgenden Text des Leitprogramms gegeben.

Der nächste Streich folgt sogleich: die StoreWord Instruktion. Die Adressberechnung mit dem ersten Register-Operand aus den Bits 20-16 und der Offset-Konstante aus Bit 15-0 ist dieselbe wie bei der LoadWord-Instruktion. Wir schreiben nun aber nichts in den Register-Block zurück, sondern wir müssen den Wert aus dem Register ,adressiert durch Bit 25-21, an den „Write data“ Eingang des Data Memory anlegen, um diesen mit dem Kontrollsignal „MemWrite“ zu speichern. Um den benötigten Wert aus dem Register-Block zu lesen, werden wir mit einem Multiplexer die Bits 25-21 an den „Read register 2“ Eingang des Blocks steuern. Am Ausgang „Data 2“ können wir nun den Registerwert abfangen und in den „Write data“ Eingang des Data Memory lenken. Die Schaltungserweiterungen sind in Abb. 4.4 abgebildet.

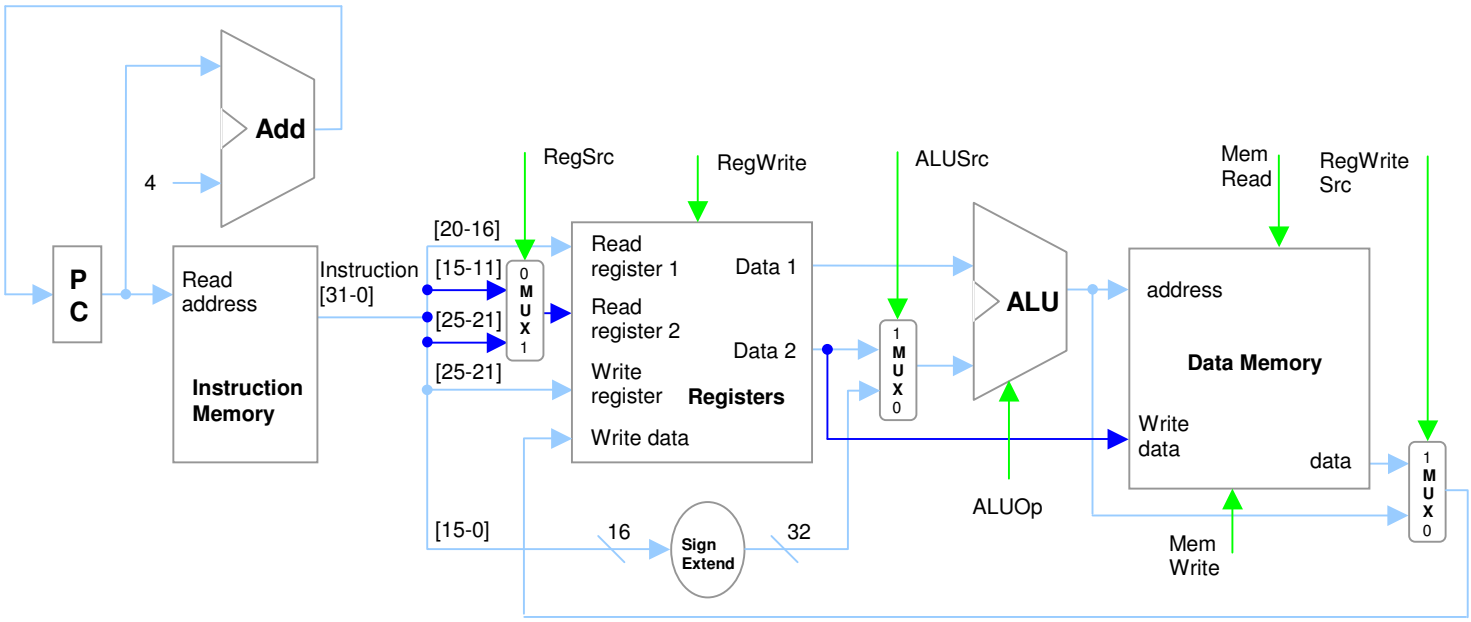


Abb. 4.4: Schaltung mit der hinzugefügten Komponente Data Memory. Die Erweiterungen für die I-Format Instruktion StoreWord sind hervorgehoben.

4.4 Branch on equal und Branch on not equal

Auch die “Branch on equal” und die “Branch on not equal”-Instruktionen sind I-Format Instruktionen. Der Wert aus Registeradresse Bit 25-21 soll mit dem Wert in Registeradresse Bit 20-16 verglichen werden. Dazu können wir den Multiplexer, den wir für die StoreWord-Instruktion benötigten, nutzen. Der Multiplexer mit dem Kontrollsignal „ALUSrc“ kann uns den Registerwert in die ALU steuern. Die ALU wird uns dann mit dem Ausgangssignal „zero“ anzeigen, ob die Werte gleich gross sind. Die Sprungadresse für einen Sprung kann nach der Sign-Extension abgezweigt werden und dann wie in Abb. 3.5 berechnet werden. Aber wann soll nun genau gesprungen werden?

Zuerst schaffen wir uns durch einen Multiplexer die Möglichkeit zwischen der normalen Programmfortsetzung, Instruktion PC+4, und der Sprungadresse auszuwählen. Nun nehmen wir an, dass aus der „Control Logic“ des Rechners ein Kontrollsignal anzeigt, ob überhaupt die Notwendigkeit für einen bedingten Sprung besteht. Nennen wir dieses Kontrollsignal „Jump“.

Betrachten wir nun die Instruktion „Branch on equal“. Sind die beiden Registerwerte identisch, so wird uns dies die ALU mit einer 1 am Ausgang „zero“ anzeigen. Ist gleichzeitig das Kontrollsignal „Jump“ auf 1 gesetzt, also bedingt gesprungen werden kann, so muss der Multiplexer die Sprungadresse an den PC-Register Eingang anlegen. Die Erweiterungen sind in Abb. 4.5 eingezeichnet.

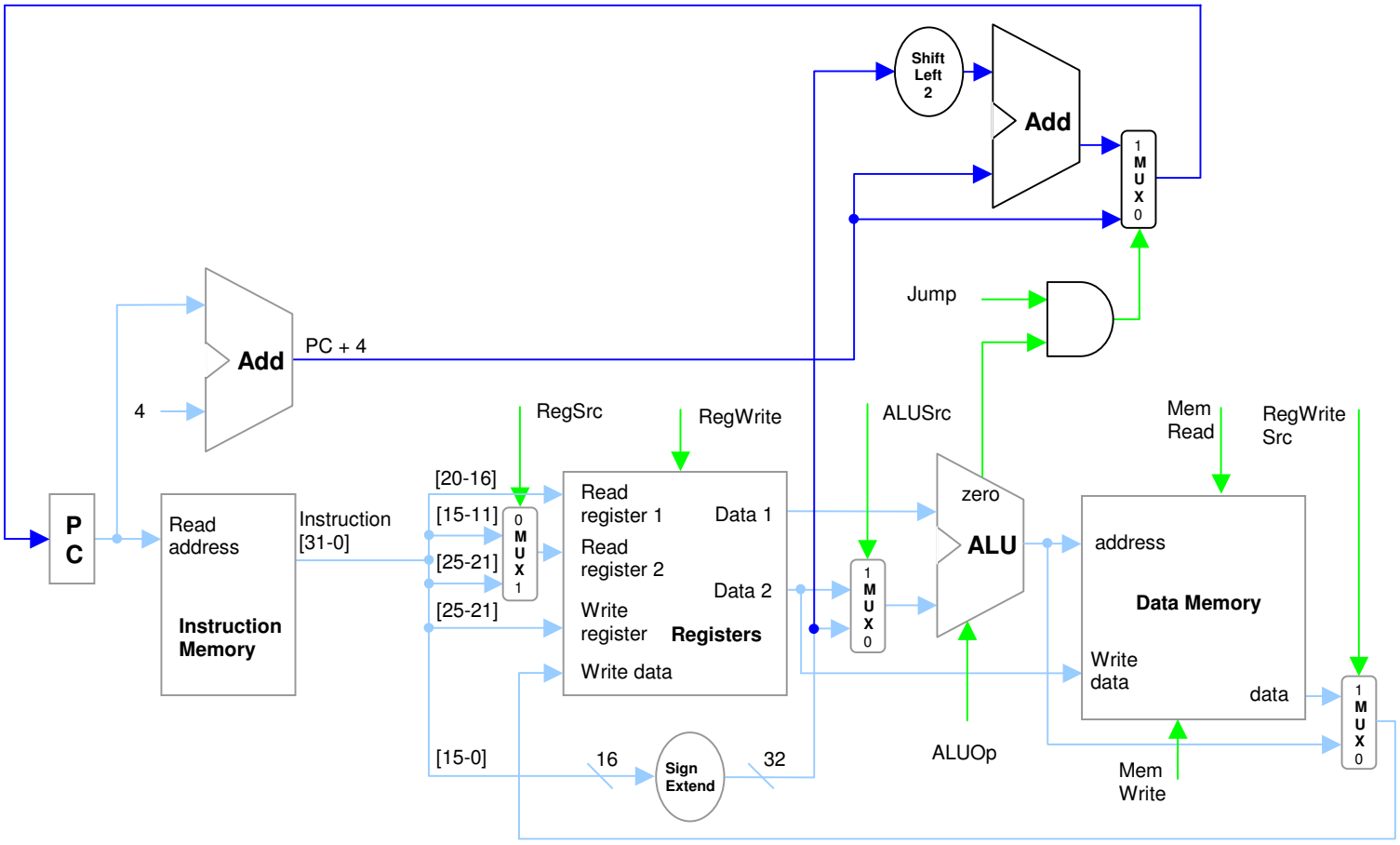


Abb. 4.5: Schaltung mit der Erweiterung für die „Branch on equal“ Instruktion. Ein Addierer und ein Shift Left 2 berechnen die Sprungadresse. Ein Multiplexer entscheidet ob die Sprungadresse oder PC + 4 ins PC-Register geschrieben wird. Ein Kontrollsignal „Jump“ zeigt die Möglichkeit an bedingt zu springen. Ist zusätzlich das ALU-Signal „zero“ auf 1, so wird der Sprung ausgeführt.

Aufgabe 4.2: Bevor Sie weiterlesen, überlegen Sie sich, welche Erweiterungen nötig werden, um eine „Branch on not equal“ Instruktion auszuführen. Sie können dafür das bis hierhin erstellte Schaltbild beliebig erweitern. Nehmen Sie dafür an, das ein Kontrollsignal anzeigt, ob bei Gleichheit (equal) oder Ungleichheit (not equal) der verglichenen Registerwerte gesprungen werden soll. Die Auflösung zu dieser Aufgabe ist im folgenden Text des Leitprogramms gegeben.

Um die Instruktion „Branch on not equal“ zu unterstützen, benötigen wir ein weiteres Signal von der Control Logic. Dieses soll uns anzeigen, ob wie bei der „Branch on equal“ Instruktion gesprungen werden soll, falls das „zero“ Signal der ALU auf 1 steht, oder ob wie bei der „Branch on not equal“ gesprungen werden soll, falls „zero“ auf 0 gesetzt ist. Durch einen Multiplexer lassen sich die Operationen unterscheiden: Bei einer „Branch on not equal“-Instruktion kann die Bedingung, ob Signal „zero“ der ALU 0 oder 1 sein muss, gesteuert werden. Nennen wir das Kontrollsignal „BranchCond“ Abb. 4.6 zeigt das Schaltbild dazu.

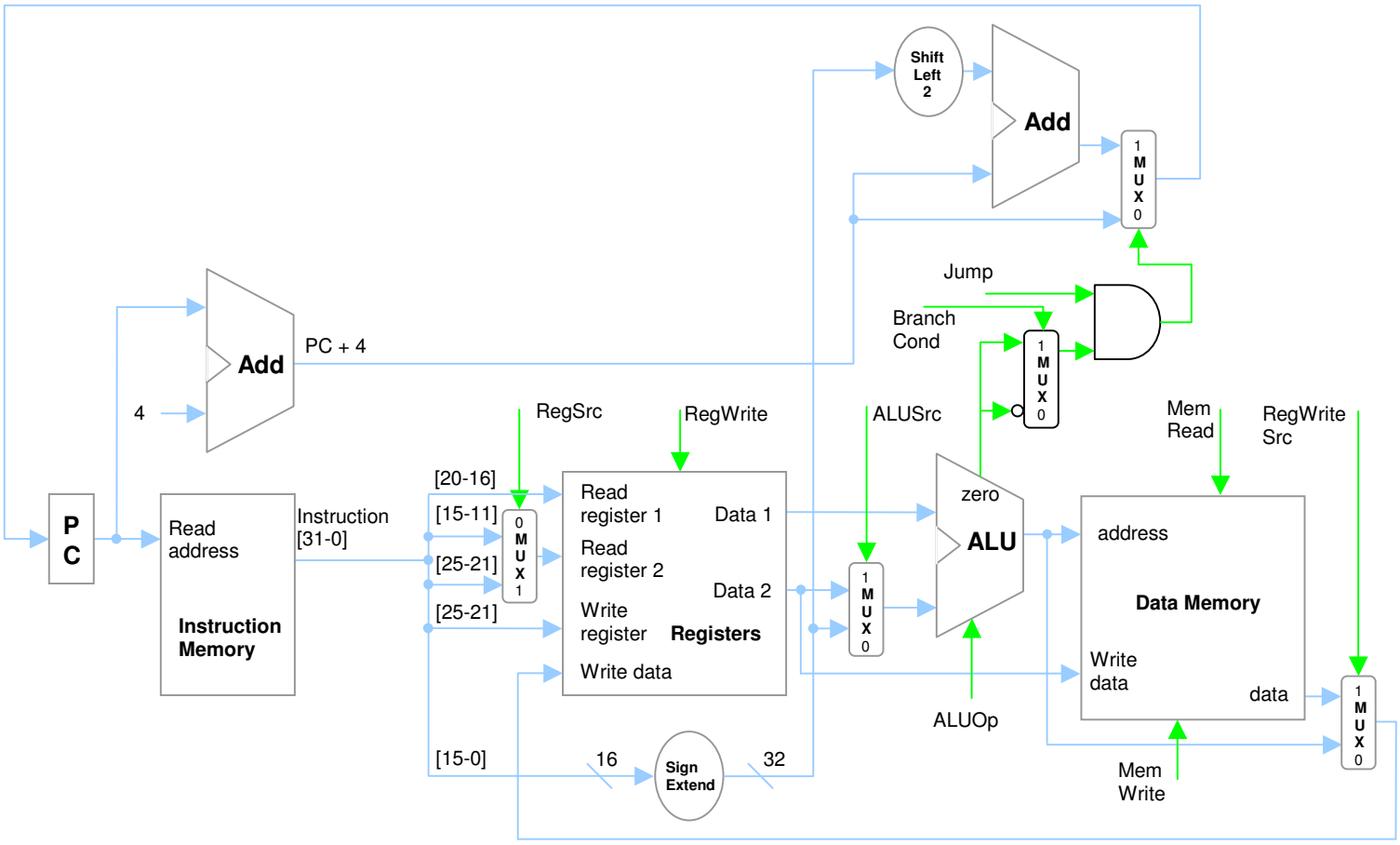


Abb. 4.6: Schaltung mit der Erweiterung für die „Branch on not equal“ Instruktion. Ein Multiplexer kann das normale oder das umgekehrte „zero“ Signal aus der ALU durchleiten. Das AND wird nun zu 1 ausgewertet, wenn das Kontrollsignal „BranchCond“ 0 ist, sowie das Signal „zero“ durch eine 0 die Ungleichheit der verglichenen Werte anzeigt.

4.5 Die J-Format Instruktionen

Als nächstes wollen wir uns zuerst der J-Format Instruktion „Jump“ widmen. Hier wird die Sprungadresse mit den Bits 25-0 gemäss der Abb. 3.2 berechnet und über einen Multiplexer in das PC-Register geleitet. Wieder ist ein neues Kontrollsignal von Nöten, taufen wir dieses „PCSrc“. Die nötigen Erweiterungen an unserer Rechnerschaltung sind in Abb. 4.7 eingezeichnet.

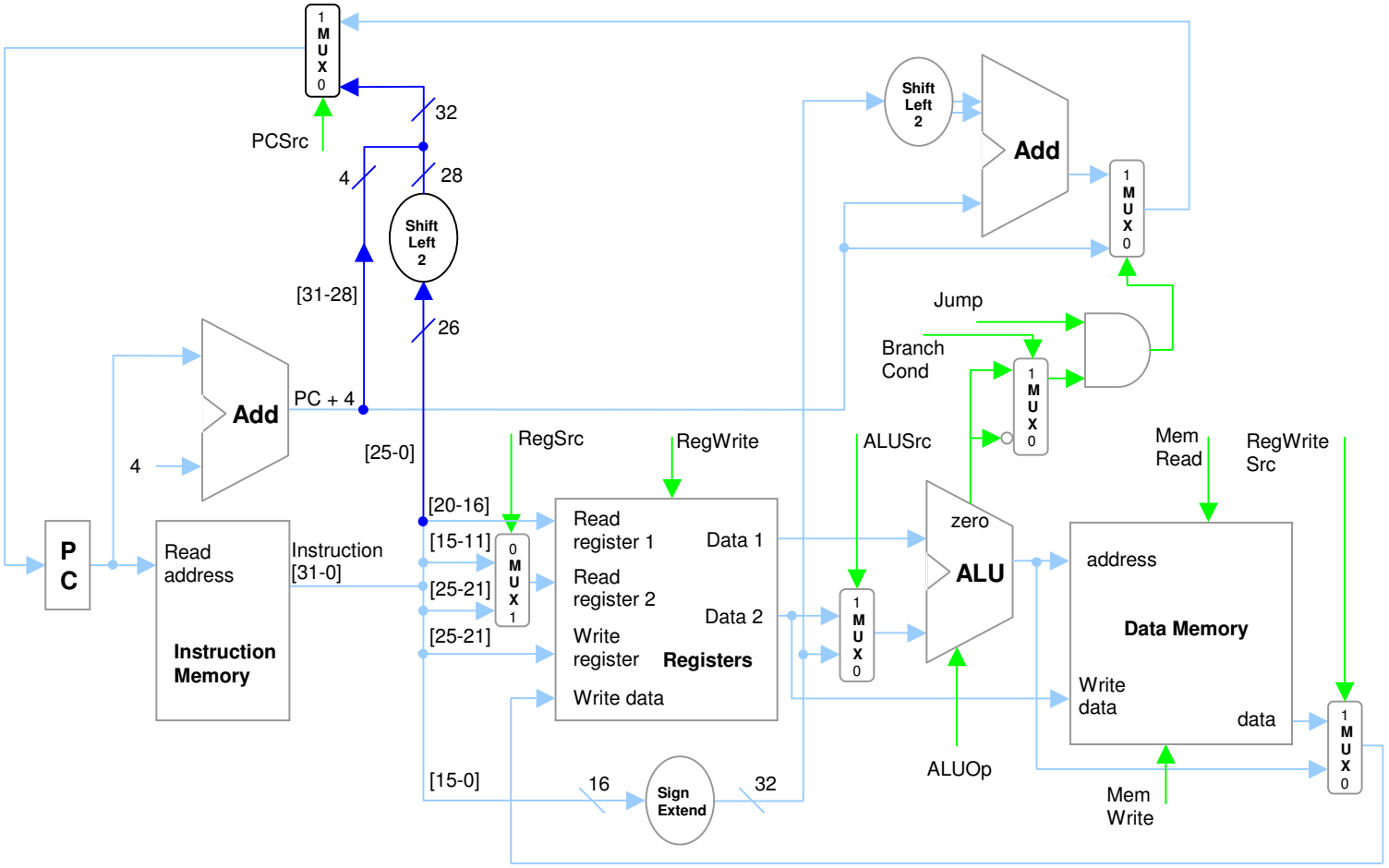


Abb. 4.7: Die Erweiterung für die Instruktion Jump. Ein Multiplexer mit Kontrollsignal „PCSrc“ steuert die für eine Jump-Instruktion berechnete Adresse zur Speicherung in das PC-Register.

Aufgabe 4.3: Bevor Sie weiterlesen, überlegen Sie sich wieder, welche Erweiterungen nötig werden, um eine „Jump and Link“ Instruktion auszuführen. Sie können dafür das bis hierhin erstellte Schaltbild beliebig erweitern. Die Auflösung zu dieser Aufgabe ist im folgenden Text des Leitprogramms gegeben.

Für eine „Jump and Link“ Instruktion ist die Funktionalität für den Sprung auf eine Adresse durch die Leitungen der „Jump“ Instruktion gegeben. Es ist also noch der „Link“ umzusetzen, also das Speichern der nächsten Instruktionsadresse PC + 4 im $ra Register. Die fixe Zielregisteradresse kann durch einen Multiplexer am „Write register“ Eingang angelegt werden. Mit einem weiteren Multiplexer müssen wir noch den zu speichernden „PC+4“-Wert am „Write data“ Eingang des Register-Blocks auswählbar machen. Dieser Schritt ist in Abb. 4.8 hervorgehoben worden. Das Kontrollsignal haben wir „WritePC“ genannt und kann für beide neuen Multiplexer verwendet werden.

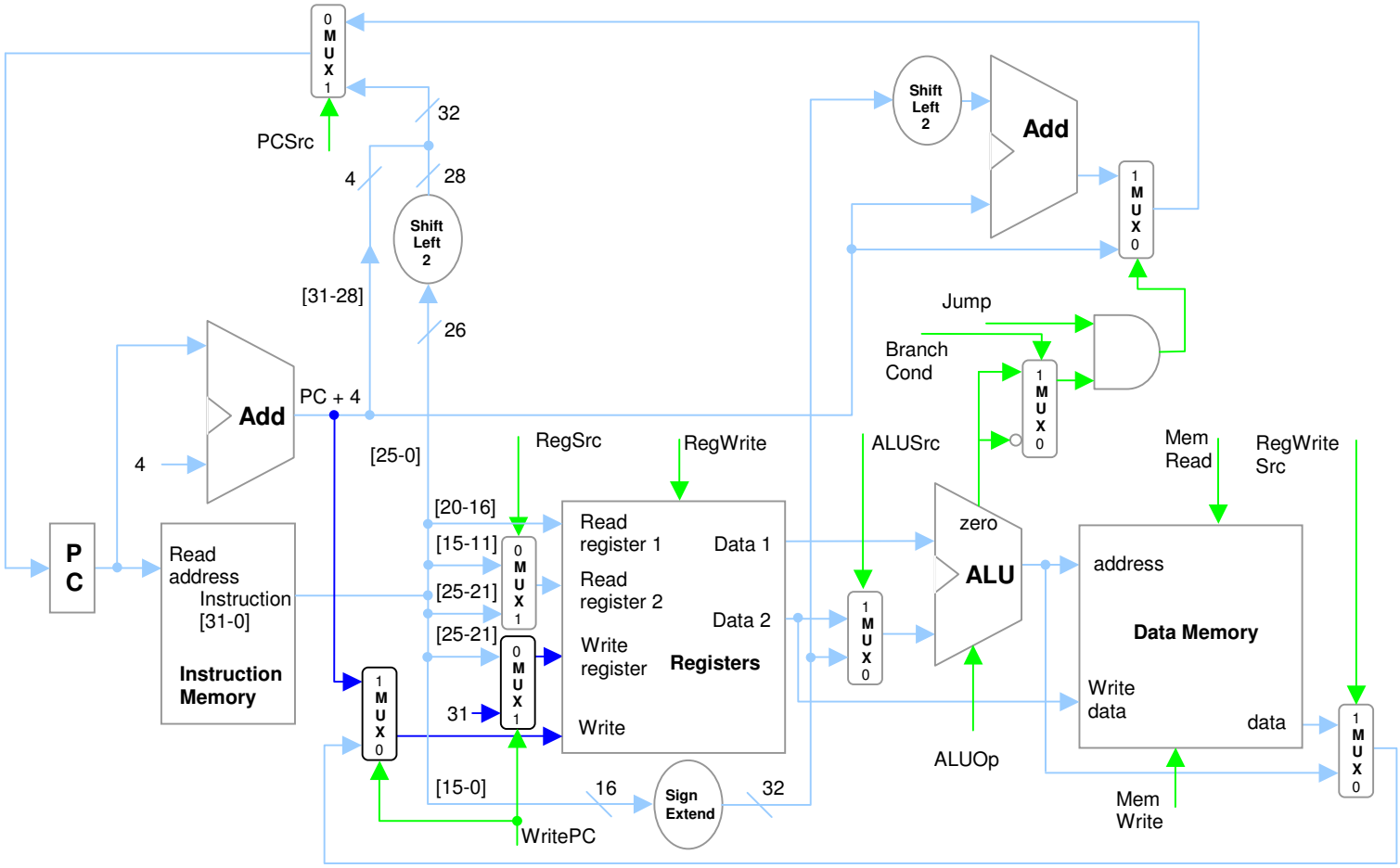


Abb. 4.8: Die Erweiterung für die Instruktion Jump and Link. Ein Multiplexer mit Kontrollsignal „WritePC“ erlaubt die Instruktionsadresse PC+4 an den Register-Block Eingang „Write data“ anzulegen und setzt die Zielregister-Adresse am Eingang „Write register“ auf 31 ($ra).

4.6 Die „Jump-Register“ Instruktion

Die Maschinencode Instruktion des Befehls „Jump Register“ ist eine Instruktion der Klasse R-Format. In der Registeradresse mit Bits 25-21 ist ein Register angegeben, dessen Wert in das PC-Register gespeichert werden soll. Die weiteren zwei Register-Adressen des R-Formates finden keine Verwendung. Wir können mit dem Multiplexer „RegSrc“ die Bits in den Register-Block Eingang „Read register 2“ eingeben. Am Ausgang „Data 2“ kann dann der Wert abgezweigt werden, und mit einem neuen Multiplexer in den Leiter zum PC-Register geleitet werden. Ein neuer Multiplexer, ein neues Kontrollsignal. Benennen wir es „JumpReg“. In Abb. 4.9 sind die nötigen Erweiterungen hervorgehoben, in Abb. 4.10 ist nochmals die gesamte bisher zusammengebaute Rechnerstruktur abgebildet.

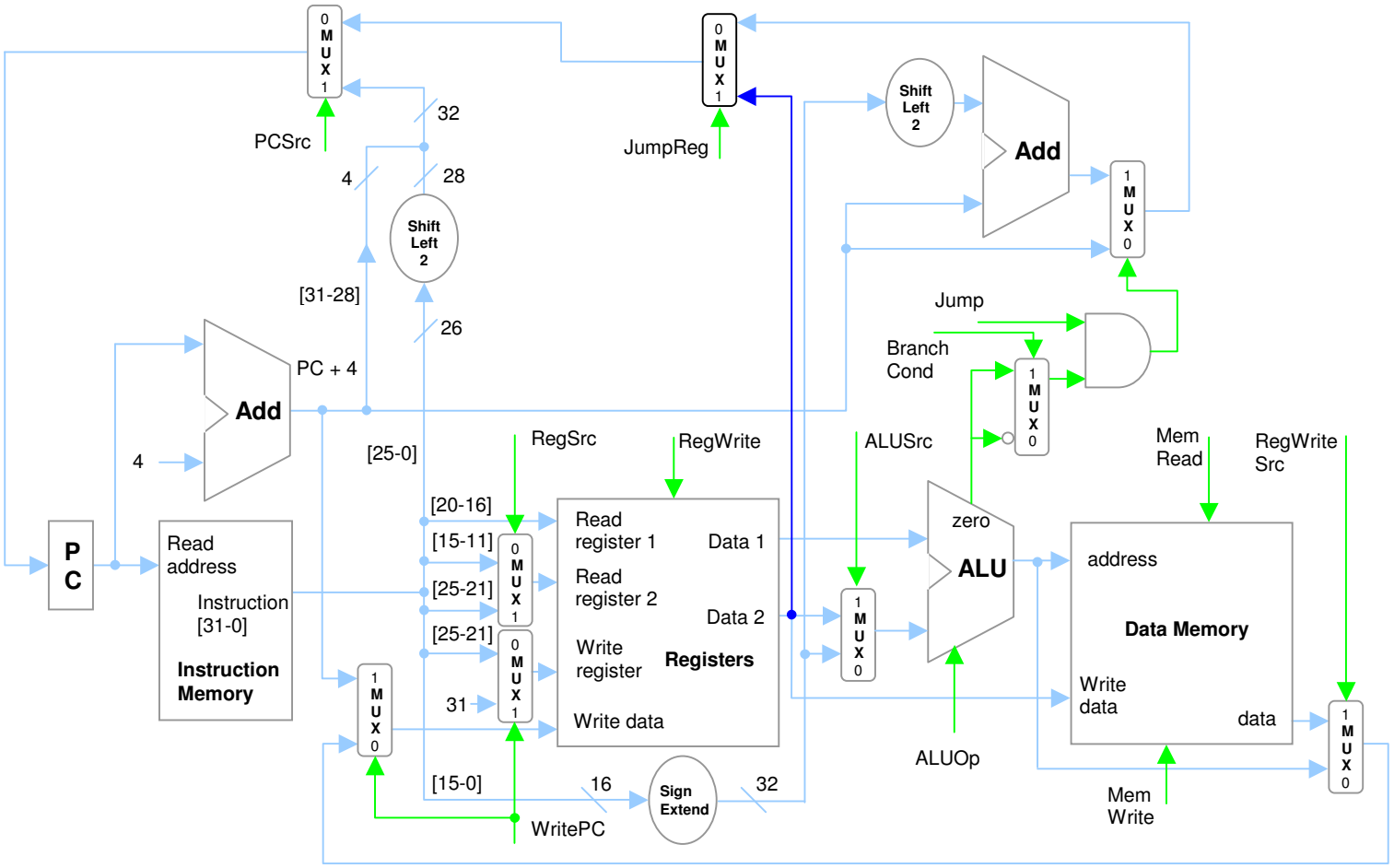


Abb. 4.9: Die Erweiterung für die Instruktion „Jump Register“. Die Sprungadresse wird am „Data 2“ Ausgang aus dem Register-Block ausgelesen und über einen Multiplexer in eine Leitung zum PC-Register geleitet. Dafür muss aber auch das Kontrollsignal „PCSrc“ auf 0 geschaltet sein.

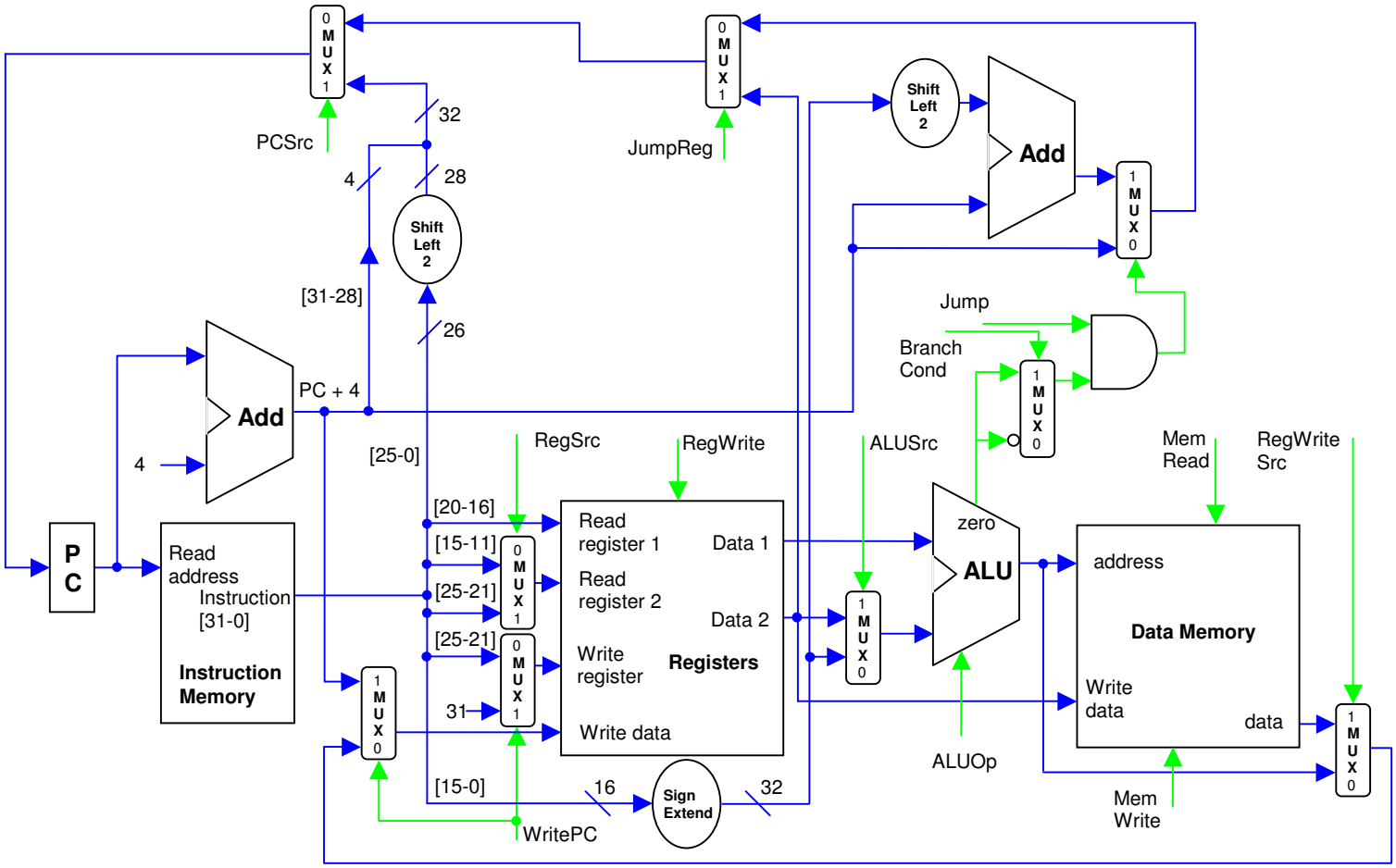


Abb. 4.10: Schaltbild der Rechnerstruktur mit den kompletten Datenpfaden. Der Rechner unterstützt das gesamte Instruction Set aus Kapitel 2 und 3.

Aufgabe 4.4: Kopieren Sie das Schaltbild für den Rechner aus Anhang A.8 5-10 mal. Zeichnen Sie nun für von Ihnen ausgewählte Instruktion aus dem Instruction Set in einem Schaltbild die verwendeten Datenleitungen ein. Falls kein Kopierer zur Verfügung steht benutzen Sie das Schaltbild aus dem Anhang A.8 und zeichnen die Datenleitungen mit Bleistift ein.

Zusammenfassung

Wir haben in diesem Kapitel unseren Rechner aus der Sicht der Datenleitungen entwickelt. Als Aussgangslage haben eine Schaltung für die R-Format ALU-Instruktionen benutzt, und diese stetig mit den Datenleitungen erweitert, die für eine zusätzlich ausführbare Instruktion nötig sind. Dabei haben wir diverse Komponenten eingebaut, die z.B. für Adressberechnungen, den Data Memory-Zugriff, aber meist für die Steuerung der Datensignale benutzt werden. Am Schluss haben wir ein komplettes Schaltbild des Rechners erhalten, in dem alle Datenleitungen richtig angeschlossen wurden. Wir haben aber auch alle Kontrollsignale an die Komponenten definiert. Um die Daten für eine Instruktion durch die richtigen Komponenten zu leiten, müssen noch die Kontrollsignale gemäss dem Opcode richtig gesetzt werden. Diese Kontrollsteuerung heisst „Control Logic“. Wir werden eine Control Logic für unseren Rechner in Kapitel 5 entwickeln.

Lernziele

Haben Sie die folgenden Lernziele des Kapitels erfüllt? Falls Sie sich nicht ganz sicher fühlen, schlagen Sie die Punkte nochmals nach. Fühlen Sie sich beim Umgang mit dem Stoff sicher? Dann bitte weiter zum Kapiteltest.

- Sie wissen, wie man eine bestehende Schaltung erweitern kann, damit neue Funktionen ausführbar werden

- Sie können in einem Schaltbild die durchlaufenen Datenpfade für eine gegebene Instruktion identifizieren

Kapiteltest 4

Aufgabe 1

Zeichnen Sie auf den Schaltbild-Vorlagen die benötigten Datenpfade ein, die bei den folgenden Instruktionen durchlaufen werden.

a) „Jump and Link“
b) „StoreWord“
c) „Jump-Register“

Vergessen Sie nicht auf ihrer Schaltbild-Vorlage die Aufgaben-Nummer und ihren Namen anzugeben.

Aufgabe 2

Zeichnen Sie auf den Schaltbild Vorlagen die benötigten Datenpfade ein, die bei den folgenden Assembler-Befehlen durchlaufen werden. Zeichnen Sie auch die Werte der Datenleitungen ein, wenn angenommen wird, das in den Registern $t0 bis $t7 die Werte 23, 756, 12, 89, 450, 0, 12, 67 gespeichert sind.

a) slt $s0, $t7, $t3
b) subi $s1, $t1, 342
c) beq $t6, $t2, 36

Vergessen Sie nicht auf ihrer Schaltbild-Vorlage die Aufgaben-Nummer und ihren Namen anzugeben.

Kapitel 5: Kontrollpfade

Einleitung

In Kapitel 4 haben wir das Schaltbild unseres Rechners mit sämtlichen nötigen Datenleitungen für das Instruction-Set versehen. Dabei haben wir einige Komponenten eingebaut, die mit einem Kontrollsignal je nach Instruktion gesteuert werden müssen. Wir entwickeln nun in diesem Kapitel noch die letzte grosse Komponente für unseren Rechner: die „Control Logic“.

1 In ersten Abschnitt erstellen wir uns eine Tabelle die aufzeigt, wie die Kontrollsignale bei einem Befehl gesetzt werden müssen. Die „Control Logic“-Komponente nimmt dabei als Input den Opcode sowie das R-Format Feld „funct“.

2 Im zweiten Abschnitt können wir uns aus der Tabelle eine logische Schaltung entwickeln, die dann als unsere Control Logic im Rechner eingebaut werden kann .

3 In Abschnitt 3 wird dann die Control Logic im Rechner eingebaut. Es werden in diesem Abschnitt auch noch 2 Beispiele aufgeführt, wie ein Befehl nun vollständig in der Rechnerstruktur abgearbeitet wird

5.1 Die Kontrollsignale

Wir wollen uns nun zuerst eine Tabelle erstellen, die uns für jeden Befehl die nötigen Kontrollsignale aufzeigt. Dafür zuerst eine Liste aller definierten Kontrollsignale in Abb. 5.1.

Kontrollsignal	Funktion
RegWrite	Es wird ein Wert in ein Register geschrieben
ALUSrc	Leitet als zweiten Operanden entweder einen Registerwert oder die Instruktionskonstante aus Bit 15-0 zur ALU
MemRead	Es wird ein Wert aus dem Data Memory gelesen
MemWrite	Es wird ein Wert in das Data Memory geschrieben
RegWriteSrc	Es wird entweder das Resultat aus der ALU oder ein Wert aus dem Data Memory zum Register-Block zurückgeleitet.
WritePC	Falls auf 1, wird der PC+4 Wert am „Write data“ Eingang, und die Registeradresse 31 am „Write register“ Eingang des Register-Blocks angelegt.
RegSrc	Speist die Registeradresse in Bit 25-21 oder die Registeradresse in Bit 15-11 in den „Read register 2“ Eingang des Register-Blocks ein
ALUOp	Operationscode der ALU bestehend aus dem Kontrollsignal Subtraction und dem 2bit breiten Operationssignal. Siehe Abb. 1.21
BranchCond	Unterscheidet „Branch on equal“ und „Branch on not equal“
Jump	Signalisiert einen unbedingten Sprung
JumpReg	Leitet die Sprungadresse des Befehls „Jump Register“ zum PC-Register
PCSrc	Leitet die Sprungadresse des Befehls „Jump“ oder des Befehls „Jump and Link“ zum PC-Register

Abb. 5.1: Tabelle aller Kontrollsignale. Im zweiten Feld ist die Funktion angegeben, die das Kontollsignal hat.

Wir können nun für jede Instruktion aus dem Instruction Set angeben, welche Kontrollsignale auf 1 oder 0 gesetzt sein müssen, damit die Daten durch die richtigen Leitungen fliessen. Spielt die Stellung eines Kontrollsignales keine Rolle, so markieren wir dessen Eintrag mit einem „X“.

Betrachten wir zum Beispiel die Instruktion LoadWord: Im Schaltbild können wir uns die benötigten Datenleitungen einzeichnen, und sehen dabei wie in Abb. 5.2, welche Komponenten durchlaufen werden.

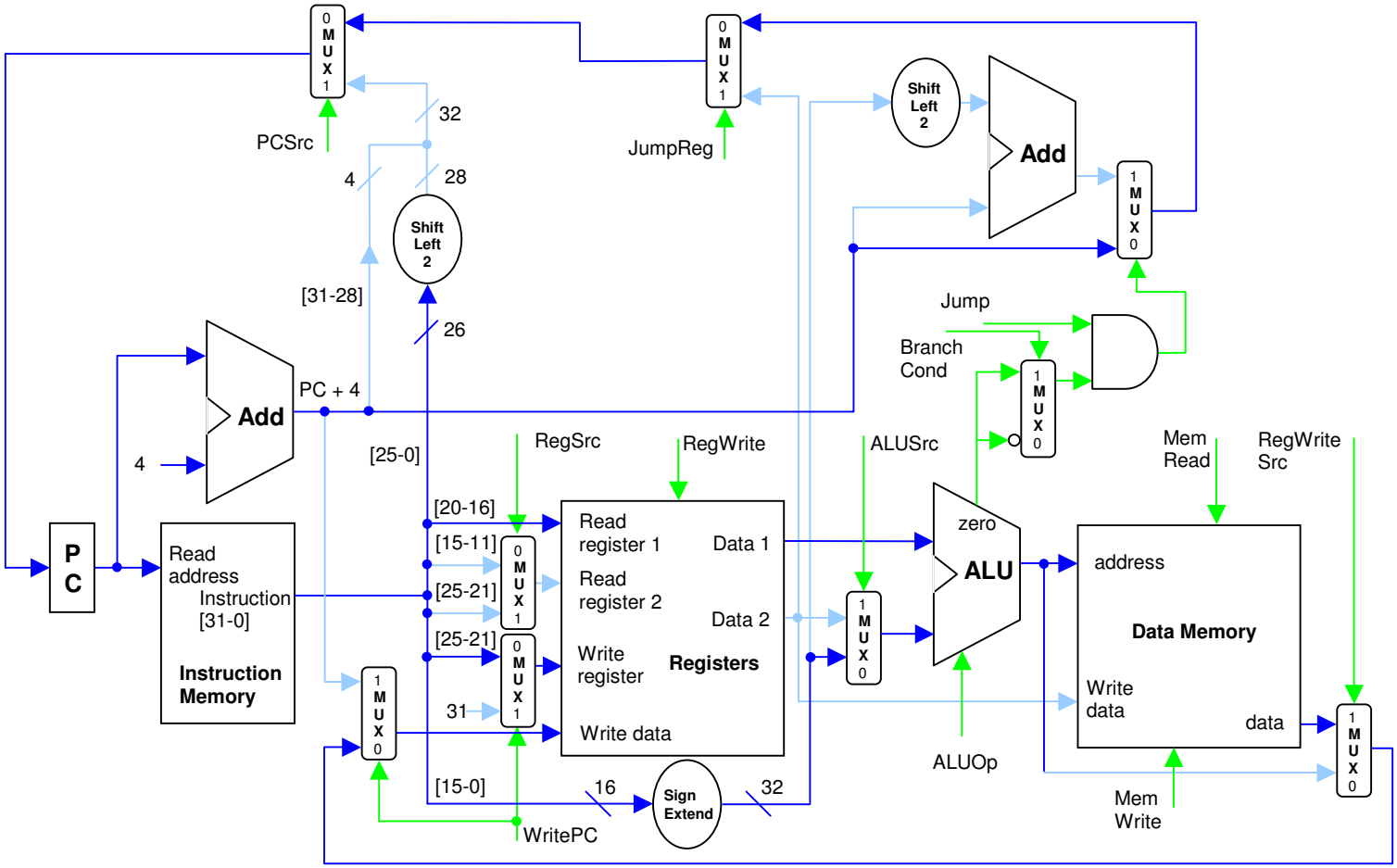


Abb. 5.2: Im Schaltbild sind die benutzten Datenleitungen für einen LoadWord Befehl hervorgehoben.

In Abb. 5.2 sehen wir, dass ein Wert am „Write data“ Eingang in den Register-Block geschrieben werden soll. Das Kontrollsignal „RegWrite“ muss also auf 1 gesetzt sein.

Das Kontrollsignal des Multiplexer „ALUSrc“ muss auf 0 gesetzt sein, um die Konstante aus den Bits 15-0 in die ALU zu leiten. Die ALU wiederum soll eine Addition ausführen, dazu legen wir die Werte 100 an das 3 Bit breite Kontrollsignal „ALUOp“ an.

Nun zum Data Memory: hier soll ein Wert aus dem Memory gelesen werden: Dies kann mit dem Kontrollsignal „MemRead“ gleich 1 erreicht werden. Gleichzeitig muss aber das Kontrollsignal „MemWrite“ auf 0 gesetzt sein. Um dem Wert aus dem Data Memory in den Register-Block zurückzuleiten, muss das Kontrollsignal des Multiplexers „RegWriteSrc“ eine 1, und das Kontrollsignale des Multiplexers „WritePC“ eine 0 führen.

Als nächstes zum Programm-Counter. 3 Multiplexer sorgen hier, dass die nächste Instruktion mit Adresse PC+4 in das PC-Register geleitet wird. Bei allen Multiplexern muss eine 0 anliegen, um das Gewünschte zu erhalten. Die Multiplexer „PCSrc“ und „JumpReg“ werden dabei direkt von der ControlLogic gesteuert, während der dritte Multiplexer mit dem Kontrollsignal „Jump“ gleich 0 gezwungen werden kann, eine 0 als Kontrollsignal zu empfangen.

Die Kontrollsignale „BranchCond“ und „RegSrc“ werden nicht durchlaufen und es spielt deshalb keine Rolle, welchen Wert an das Kontrollsignal führt.

Die Kontrollsignale für eine LoadWord Instruktion sind in Abb. 5.3 nochmals zusammengefasst.

Befehl	Opcode	funct	Reg Write	ALU Src	Mem Read	Mem Write	RegWrite Src	Write PC	Reg Src	ALU Op	Branch Cond	Jump	JumpReg	PCSrc
lw	13	-	1	0	1	0	1	0	x	100	x	0	0	0

Abb. 5.3: Tabelle aller Kontrollsignale für den Befehl LoadWord.

Mit dem gleichen Vorgehen können wir uns die Kontrollsignale zu jeder Instruktion des Instruction Set bestimmen.

Die Tabelle in Abb. 5.4 fasst die benötigten Kontrollsignale zu jeder Instruktionen zusammen.

Befehl	Opcode	funct	Reg Write	ALU Src	Mem Read	Mem Write	RegWrite Src	Write PC	Reg Src	ALU Op	Branch Cond	Jump	JumpReg	PCSrc
add	32	4	1	1	0	0	0	0	0	100	x	0	0	0
sub	32	5	1	1	0	0	0	0	0	101	x	0	0	0
and	32	0	1	1	0	0	0	0	0	000	x	0	0	0
or	32	2	1	1	0	0	0	0	0	010	x	0	0	0
slt	32	7	1	1	0	0	0	0	0	111	x	0	0	0
addi	8	-	1	0	0	0	0	0	x	100	x	0	0	0
subi	9	-	1	0	0	0	0	0	x	101	x	0	0	0
andi	10	-	1	0	0	0	0	0	x	000	x	0	0	0
ori	11	-	1	0	0	0	0	0	x	010	x	0	0	0
slti	12	-	1	0	0	0	0	0	x	111	x	0	0	0
lw	13	-	1	0	1	0	1	0	x	100	x	0	0	0
sw	14	-	0	0	0	1	x	0	1	100	x	0	0	0
beq	16	-	0	1	0	0	x	0	1	101	1	1	0	0
bne	17	-	0	1	0	0	x	0	1	101	0	1	0	0
j	6	-	0	x	0	0	x	0	x	x	x	0	0	1
jal	7	-	1	x	0	0	x	1	x	x	x	0	0	1
jr	32	8	0	x	0	0	x	0	1	x	x	0	1	0

Abb. 5.4: Tabelle aller Befehle mit den dazugehörigen Kontrollsignalen. Ein x markiert, dass es für einen Befehl keine Rolle spielt, wie das Kontrollsignal gesetzt ist, bei einer 1 oder 0 ist dieser Wert für die Instruktionsausführung zwingend.

5.2 Die Control Logic

Bestimmt durch die Instruktionsfelder „Opcode“ sowie dem Feld „funct“ für R-Format Instruktionen, müssen nun die richtigen Kontrollsignale an die Rechnerkomponenten angelegt werden. Wir gehen dabei folgendermaßen vor: Wir betrachten jedes Kontrollsignal einzeln und bestimmen, bei welchen Opcodes eine 1 (oder eine 0) angelegt werden müssen. Dazu bauen wir uns jeweils eine logische Schaltung. Am Schluss können dann einfach die logischen Schaltungen zu jedem Kontrollsignal zusammengefasst werden und fertig ist die Control Logic. Natürlich ist das Ergebnis nicht optimal, wir hoffen dafür, dass der Aufbau der Control Logic so auf einfache Weise verstanden werden kann.

Beginnen wir mit dem Kontrollsignal **MemWrite**. Hier wird nur eine 1 ausgegeben, wenn der Opcode gleich 14 ist. Die Opcode Bits 5-0 (in der Maschinencode-Instruktion die Bits [31-25]) sind also 001110 (14). Durch ein Mehrweg-AND können wir diese Bedingung garantieren. Die Schaltung kann dann wie in Abb. 5.5 aussehen.

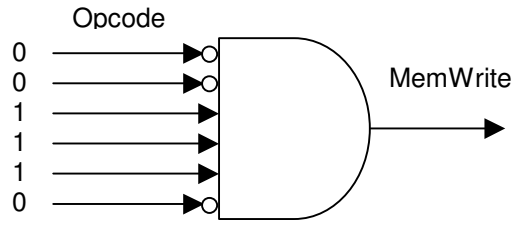


Abb. 5.5: Das Logische Mehrweg-AND wertet nur zu 1 aus, wenn der Opcode die Werte 001110 anlegt. Nur dann wird vom Kontrollsignal „MemWrite“ eine 1 erwartet.

Die Schaltungen für die Kontrollsignale „MemRead“, „WritePC“ und „JumpReg“ können auf die genau gleiche Weise erstellt werden.

Aufgabe 5.1: Zeichnen Sie die Schaltungen für die Kontrollsignale „MemRead“, „WritePC“ und „JumpReg“.

Betrachten wir nun die Kontrollsignale „BranchCond“ und „RegWriteSrc“. Hier haben wir in der Tabelle mehrer Einträge mit einem x, es ist also egal ob wir das x als eine 1 oder eine 0 annehmen. Wir füllen die x-en mit einer 0 und erhalten wieder nur jeweils eine 1 für einen bestimmten Opcode. Das Kontrollsignal „**BranchCond**“ wertet für den Opcode 16: 010000 zu einer 1 aus. Die Schaltung ist in Abb. 5.6 gezeigt.

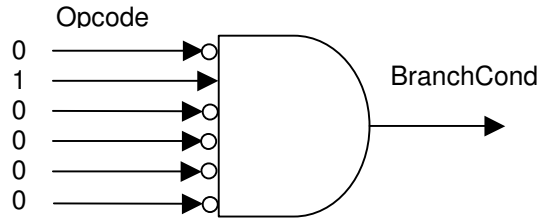


Abb. 5.6: In a) ist die logische Schaltung für das Kontrollsignal „BranchCond“ zu sehen. Die Tabelleneinträge mit einem X zum Kontrollsignal wurden als 0 angenommen.

Aufgabe 5.2: Zeichnen Sie die Schaltung für das Kontrollsignal „RegWriteSrc“. Kommt Ihnen das Ergebnis bekannt vor?

Wir widmen uns nun den Kontrollsignalen „Jump“ und „PCSrc“. Es gibt zu diesen Kontrollsignalen jeweils zwei Opcodes, die das Kontrollsignal auf 1 auswerten lassen. Für das Kontrollsignal „**Jump**“ sind dies die Opcodes 16: 010000 und 17: 010001. Die Schaltung ist aus zwei Mehrweg-AND’s aufgebaut, für jeden Opcode das Entsprechende. Mit einem Logischen OR können dann die Mehrweg-AND’s kombiniert werden, so dass bei jedem der beiden möglichen Opcodes das Signal 1 ausgegeben wird. Die Schaltung finden Sie in Abb. 5.7.

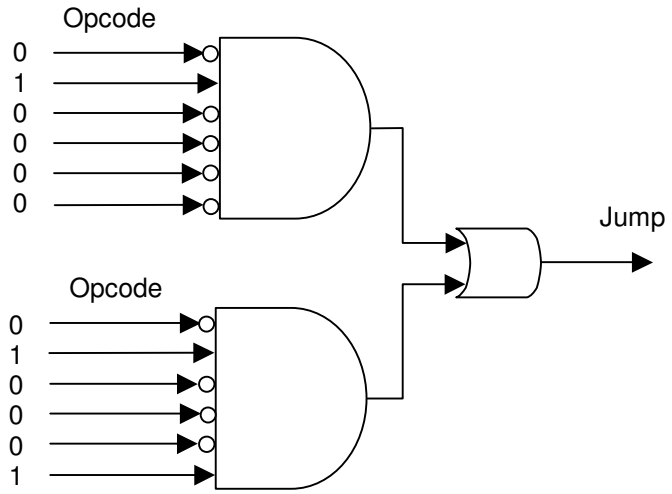


Abb. 5.7: Zeigt die Schaltung für das Kontrollsignal „Jump“ gezeigt. Die Schaltung wird aus zwei Mehrweg-AND’s für den Opcode aufgebaut, die dann mit einem OR kombiniert werden

Aufgabe 5.3: Zeichnen Sie die Schaltung für das Kontrollsignal „PCSrc“.

Betrachten wir als nächstes das Kontrollsignal „**ALUSrc**“. Füllen wir das x in der Spalte für jr mit einer 1 und die zwei anderen x-en mit einer 0, so werden alle Operationen mit dem Opcode 32 (R-Format) das Kontrollsignal zu einer 1 auswerten. Für eine Prüfung ob der Opcode gleich 32 ist, ist wieder nur die Betrachtung der Opcodeleitung mit Index [5] nötig. Zusätzlich wird das Kontrollsignal bei den Opcodes 16: 010000 und 17: 010001 zu einer 1 ausgewertet. Die Schaltung ist in Abb. 5.8 gezeigt.

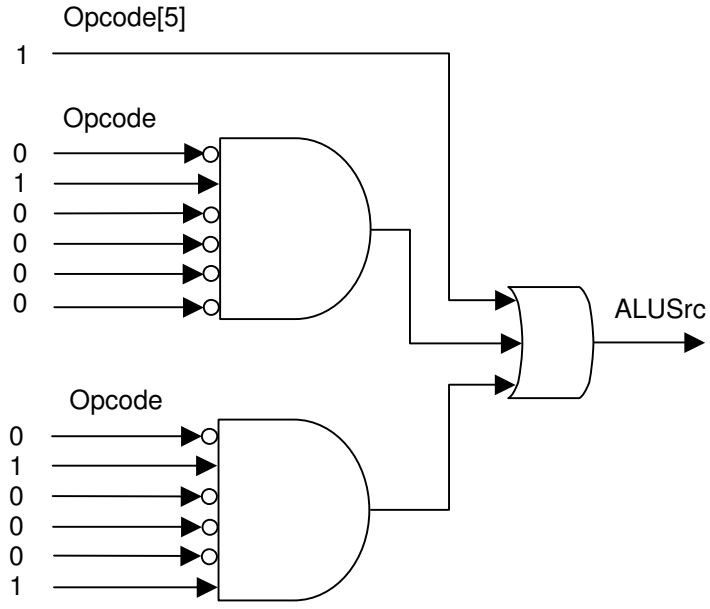


Abb. 5.8: Die Schaltung für das Kontrollsignal ALUSrc. Alle Operationen mit dem Opcode 32 werden zu 1 ausgewertet, sowie die Opcodes 16 und 17.

Als nächstes folgt die Schaltung für das Kontrollsignal „**RegSrc**“. Wir nehmen wieder alle x-en der Spalte als 0 an. Wie bei den Kontrollsignalen „Jump“ oder „PCSrc“ führen nun mehrere Opcodes zu einer 1 des Kontrollsignals. Es sind diesmal aber nicht zwei sondern 4 Opcodes: der Opcode 14: 001110, 16: 010000, 17: 010001 sowie der Opcode 32: 100000 mit dem „funct“-Feld 8: 001000. Die Schaltung ist in Abb. 5.9 gezeigt.

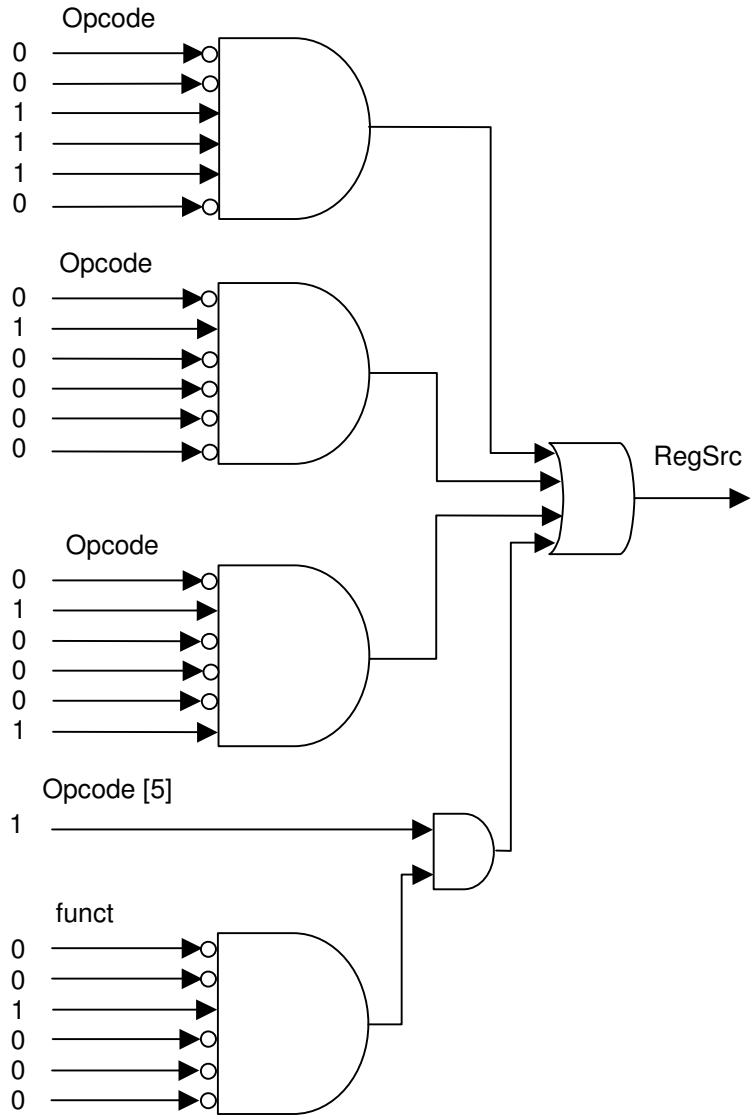


Abb. 5.9: Die Schaltung für das Kontrollsignal RegSrc. Vier verschiedene Opcodes führen zu einer 1 am Kontrollsignal.

Kommen wir zum Kontrollsignal „**RegWrite**“. In der Spalte ist zu sehen, dass das Signal immer zu einer 1 auswertet, außer bei 5 spezifischen Opcodes: 14: 001110, 16: 010000, 17: 010001, 6: 000110 und 7: 000111. Wir fassen die betroffenen Opcodes wieder mit Mehrweg-AND’s und einem Logischen OR zusammen. Nun wollen wir aber bei einem solchen Opcode keine 1 sondern eine 0 als Kontrollsignal ausgeben. Bei allen anderen Opcodes soll nun keine 0 sondern eine 1 ausgegeben werden. Dies können einfach mit der Umkehrung des Output-Signals der Schaltung erreichen. Abb. 5.10 zeigt wie es gemacht wird.

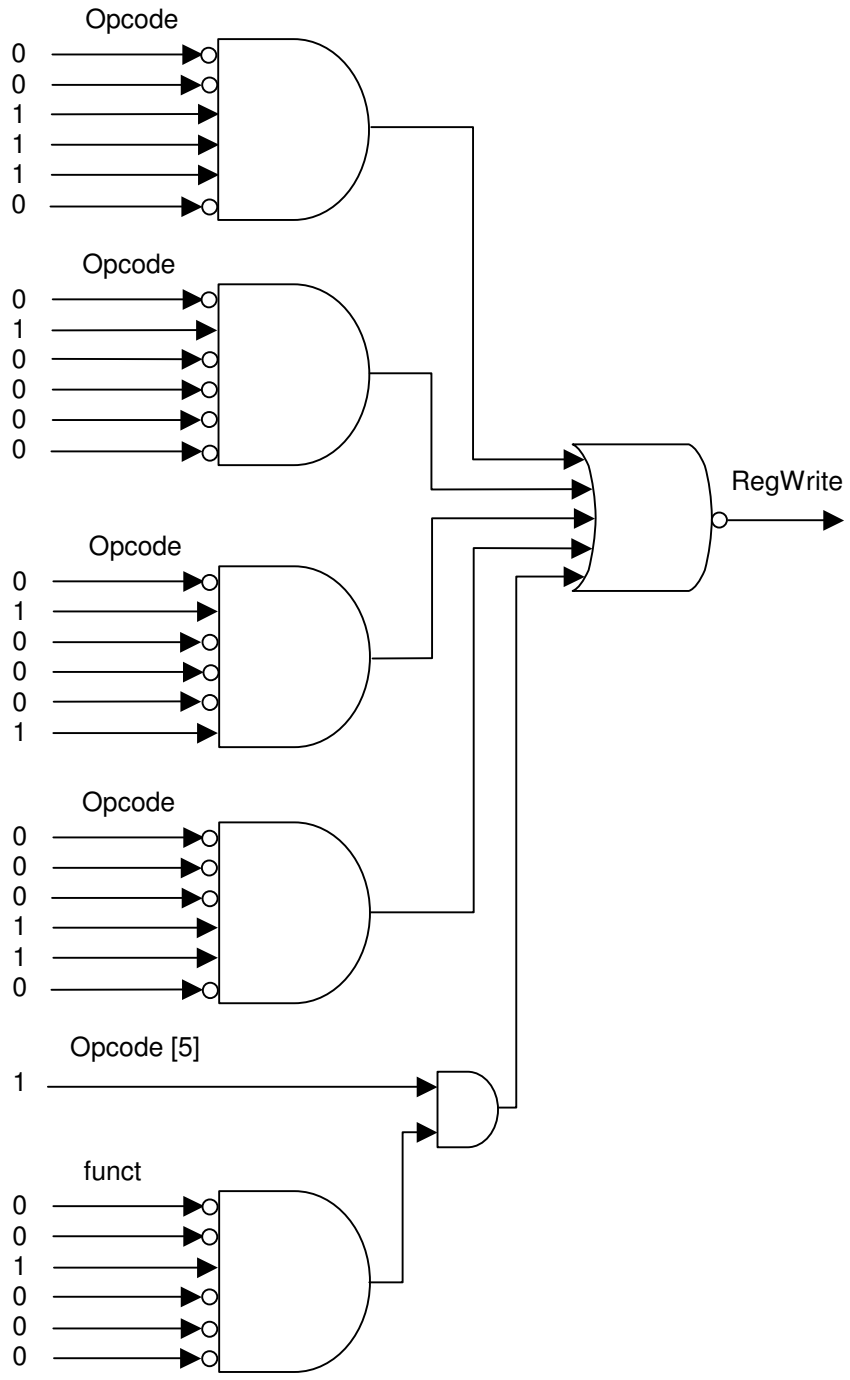


Abb. 5.10: Die Schaltung für das Kontrollsignal RegWrite. Fünf verschiedene Opcodes führen zu einer 0 am Kontrollsignal, dies wird durch ein NOT am Signal-Output erreicht. Für alle anderen Opcodes ist das Schaltungsergebnis eine 1.

Wir haben nun nur noch das 3-Bit breite Kontrollsignal „ALUOp“ in einer Schaltung umzusetzen, dies stellt aber auch den größten Brocken dar. Wir müssen nämlich für jede der 3 Signalleitungen eine eigene Schaltung umsetzen.

Betrachten wir als erstes das Kontrollsignal mit dem Index [0]. Ersetzen wir die x-en des Kontrollsignals mit Index 0 durch eine 0, so haben wir 6 verschiedene Opcodes, die das Kontrollsignal ALU-OP[0] zu einer 1 auswerten: Opcode 32: 100000 mit „funct“-Feld 5: 000101. Opcode 32: 100000 mit „funct“-Feld 7: 000111. Opcode 9: 001001. Opcode 12: 001100. Opcode 16: 010000 und Opcode 17: 010001. Die Schaltung ist in Abb. 5.11 dargestellt.

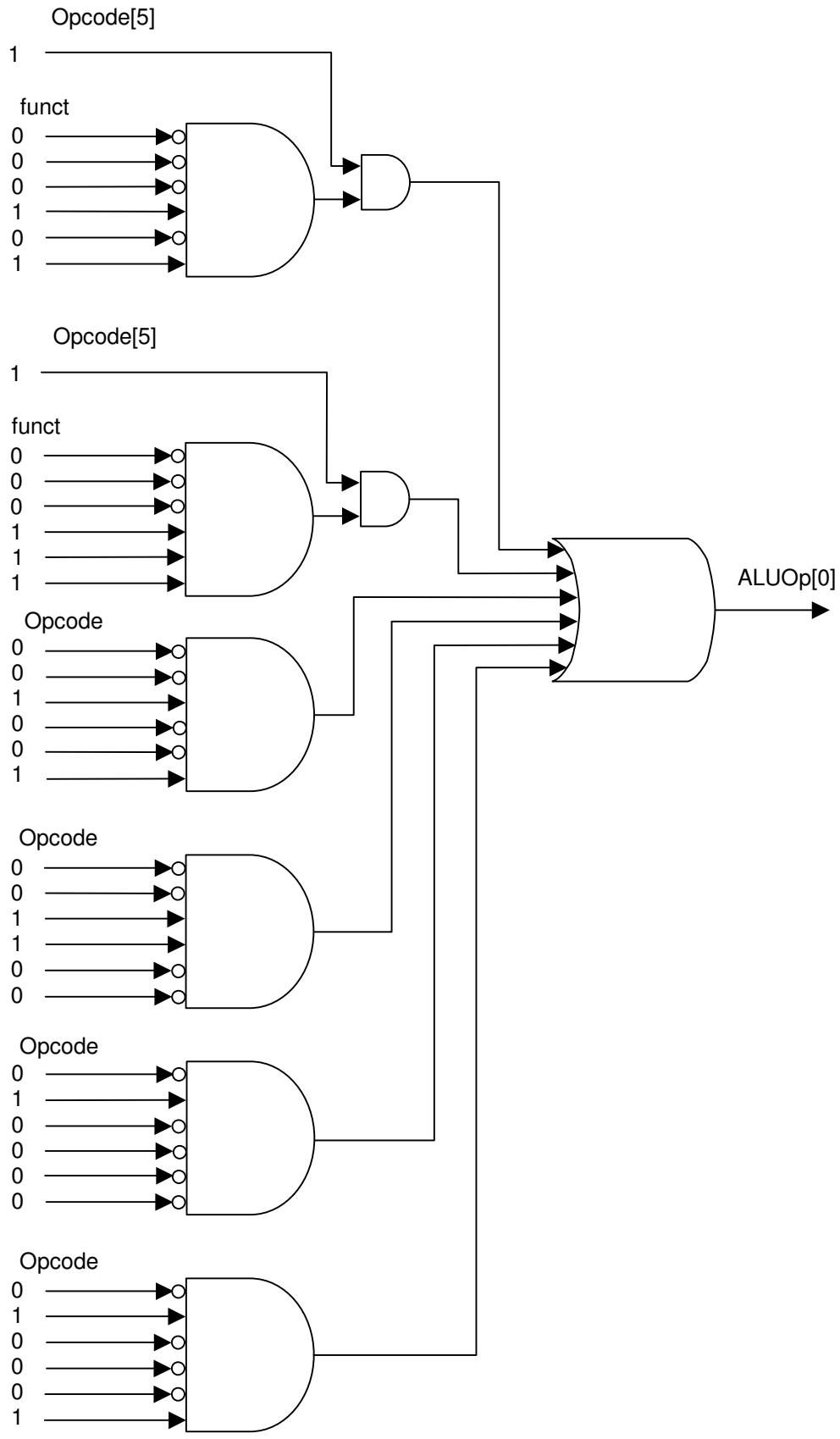


Abb. 5.11: Die Schaltung für das erste Bit des 3 Bit breiten Kontrollsignal „ALUOp“.

Kommen wir zum Index [1] des Kontrollsignals „ALUOp“: Wenn wir die x-en zu dieser Spalte als 0 betrachten, so gibt es 4 Opcodes, die diese Stelle des Kontrollsignal zu einer 1 auswerten lassen. Opcode 32: 100000 mit „funct“-Feld 2: 000010. Opcode 32: 100000 mit „funct“-Feld 7: 000111. Opcode 11: 001011. Opcode 12: 001100. Die Schaltung ist in Abb. 5.12 dargestellt.

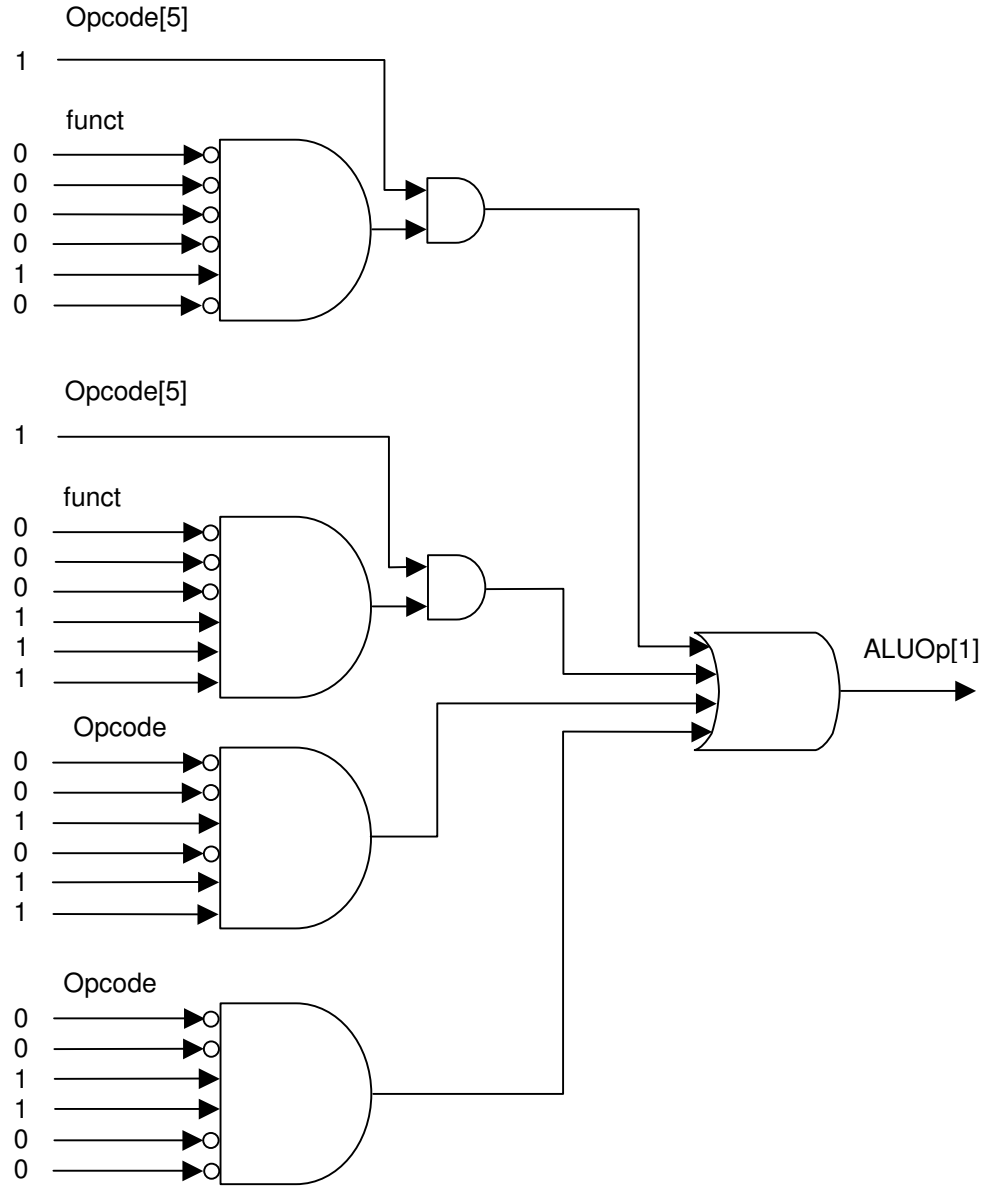


Abb. 5.12: Die Schaltung für das zweite Bit des 3 Bit breiten Kontrollsignals „ALUOp“.

Fehlt noch die Schaltung für das dritte Bit des Kontrollsignals „ALUOp“, also das Kontrollsignal mit dem Index [2]. In der Tabelle sehen wir, dass wenn die x-en der Spalte mit 1 gefüllt werden, nur 4 Opcodes das Kontrollsignal zu einer 0 auswerten lassen. Sonst wird immer eine 1 ausgegeben. Die Opcodes sind 32: 100000 mit dem „funct“-Feld 0: 000000, der Opcode 32: 1000000 mit dem „funct“-Feld 2: 000010, der Opcode 10: 001010 und der Opcode 11: 001011. Wir können also wieder den selben Trick wie für das Kontrollsignal „RegWrite“ anwenden und das Output-Signal mit einem NOT umkehren. Die Schaltung ist in Abb. 5.13 abgebildet.

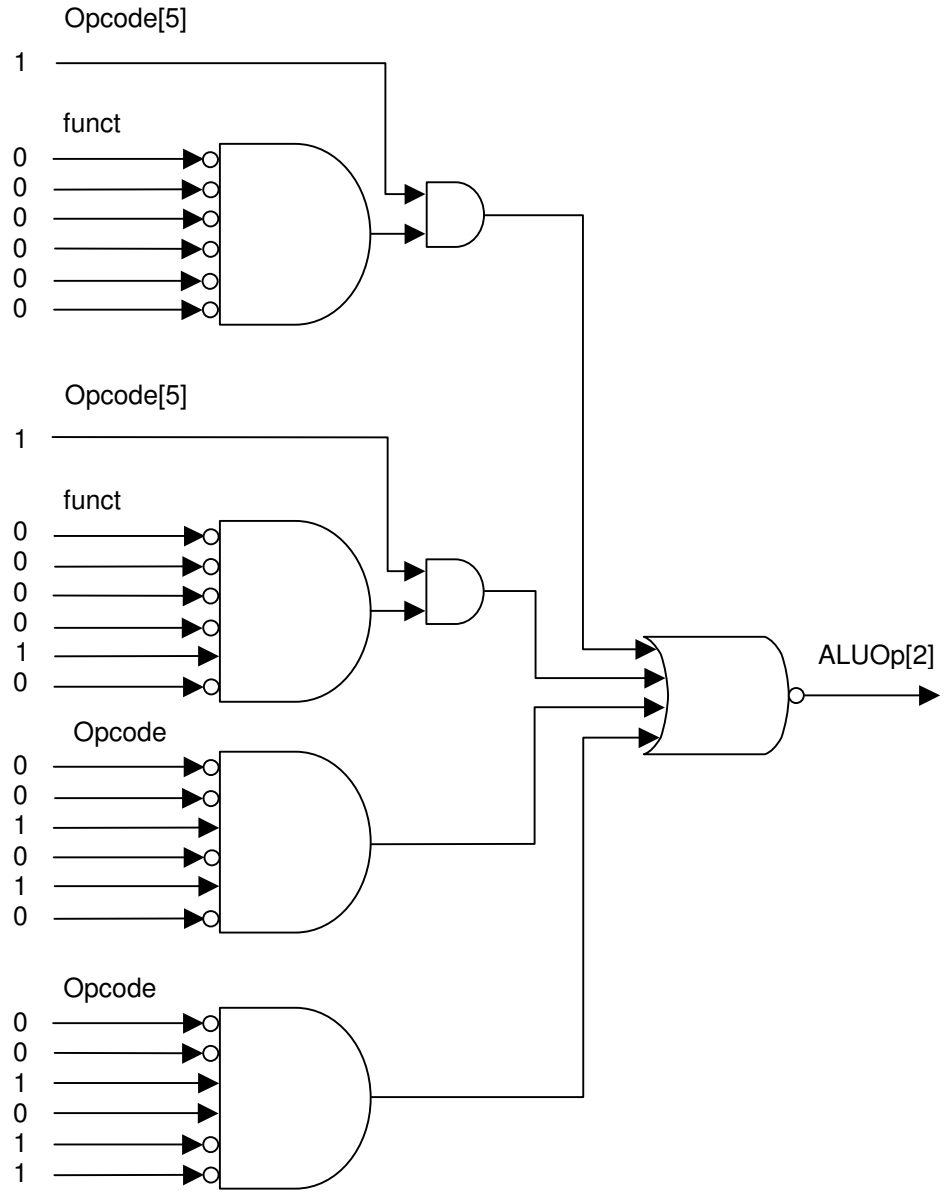


Abb. 5.13: Die Schaltung für das dritte Bit des 3 Bit breiten Kontrollsignal „ALUOp“.

Wir haben nun für jedes Kontrollsignal eine Schaltung entworfen, wobei die Schaltung für das Kontrollsignal durch den Opcode und eventuell ein „funct“-Feld gesteuert wird. Wir wollen nun all diese Kontrollsignalschaltungen in eine Komponente zusammenfassen. Diese Komponente wird dann unsere Control Logic sein. Um die Schaltungen zusammenzufassen, können wir einfach die Opcodesignale verzweigen und in jede der Kontrollsignalschaltungen führen. Dasselbe gilt auch für die Signale des Feldes „funct“. Falls das Mehrweg-AND für einen bestimmten Opcode mehrmals in den Schaltungen vorkommt, können wir das Mehrweg-AND nur einmal einfügen, und das Output-Signal in die weiteren Kontrollsignalschaltungen leiten. In Abb. 5.14 ist die komplette Control Logic abgebildet, wobei eben die Mehrweg-AND’s für einen Opcode nur jeweils einmal vorkommen.

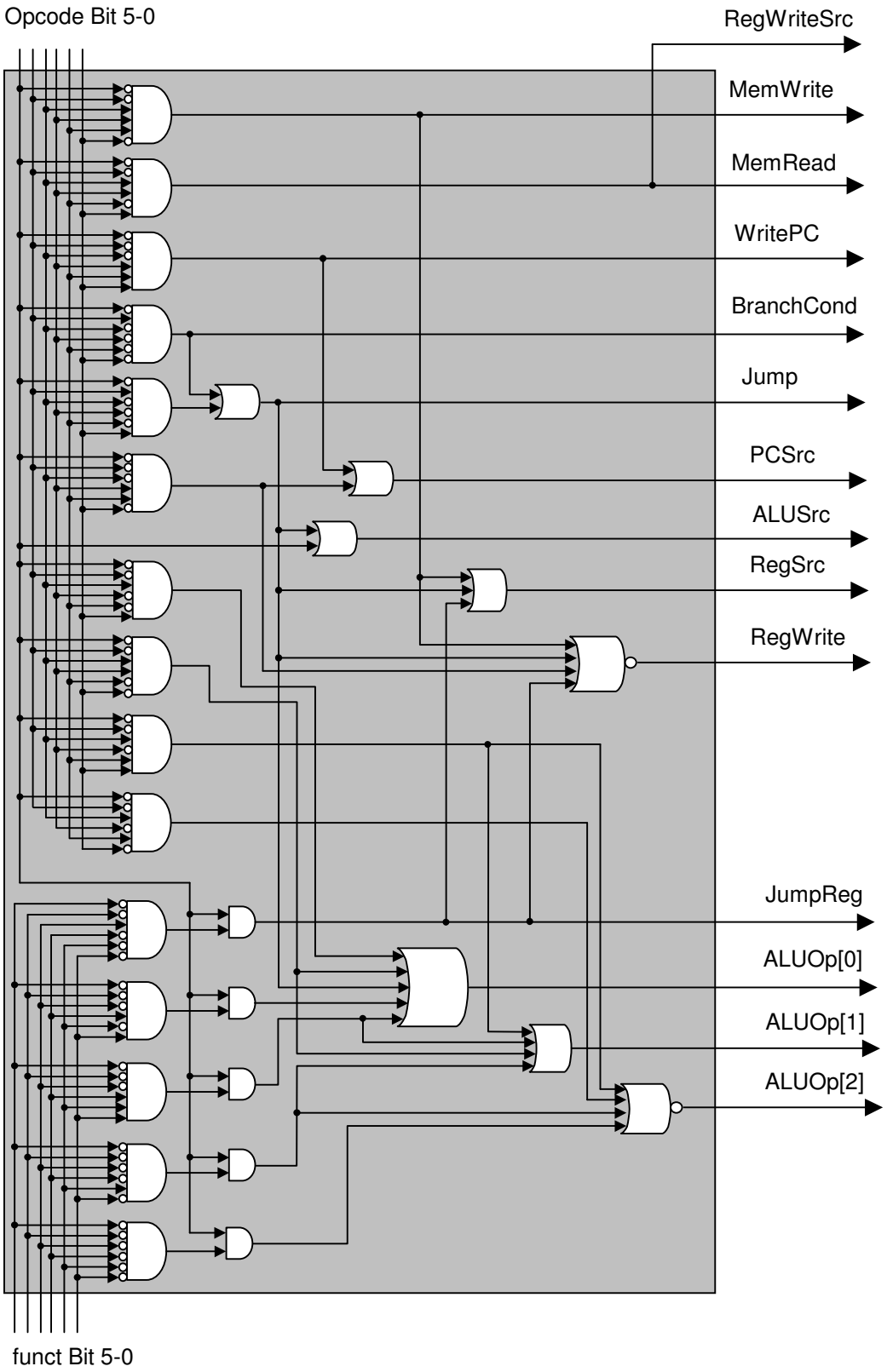


Abb. 5.14: Die vollständige Schaltung für die ControlLogic. Die Schaltung nimmt als Input den Opcode einer Instruktion, sowie die Bits des R-Format-Feldes „funct“. Ausgegeben werden sämtliche Kontrollsignale der Rechnerstruktur.

Für die gesamte Control Logic führen wir nun noch ein Schaltungssymbol ein. Dieses wird in Abb. 5.15 vorgestellt.

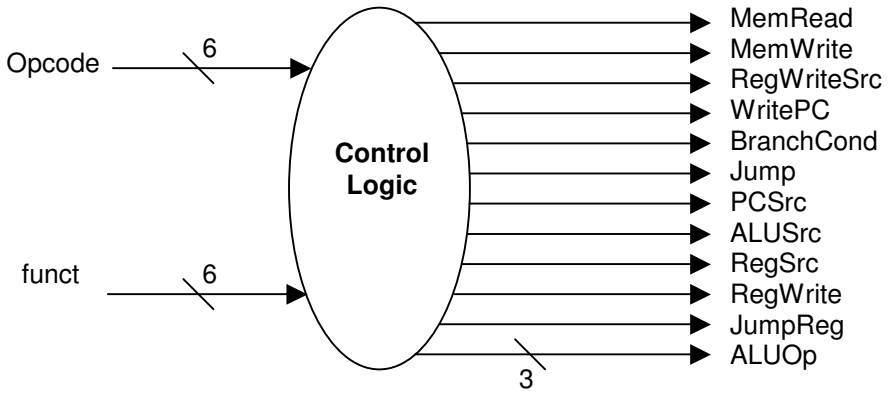


Abb. 5.15: Schaltungssymbol für die ControlLogic

Aufgabe 5.4: Kopieren Sie das Schaltbild für die Control Logic aus Anhang A.7 wieder 5-10 mal. Zeichnen Sie nun für jede der von Ihnen ausgewählten Instruktion aus dem Instruction Set in einem Schaltbild die Leitungen ein, die eine 1 als Signal tragen.

5.3 Einbau der Control Logic und zwei Beispiele

Die Control Logic in unseren Rechner einzubauen ist keine Hexerei mehr. Den Opcode finden wir jeweils in den Bits 31-26 in der Instruktion, und für das Feld „funct“ werden die Bits 5-0 zur Control Logic abgezweigt. Die Ausgänge werden dann nur noch mit den entsprechenden Komponente verbunden, und fertig ist unser Rechner. Die Abb. 5.16 zeigt das vollständige Schaltbild unseres Rechners.

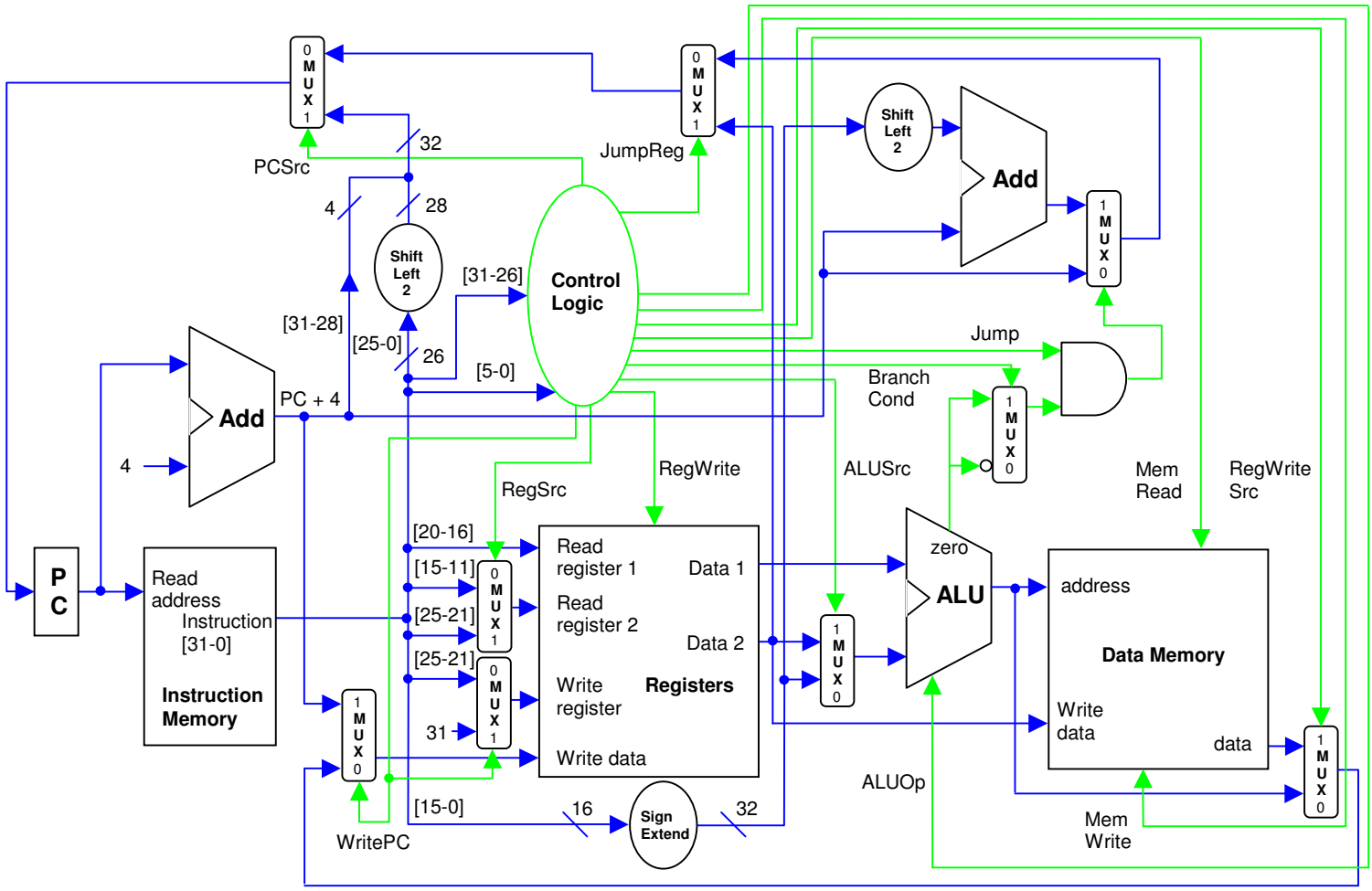


Abb. 5.16: Das vollständige Schaltbild unserer Rechnerstruktur mit allen Datenpfaden und allen Kontrollpfaden.

Wir wollen zum Kapitelabschluss noch zwei Beispiele betrachten. Dabei schauen wir jeweils in die Control Logic, und markieren mit blau alle Leitungen, die eine 1 als Signal tragen. Danach ist jeweils die Rechnerstruktur gezeigt, bei der die durchlaufenen Datenpfade blau hervorgehoben sind. Die Kontrollsignale, die eine 1 als Signal tragen sind grün hervorgehoben. Beim 3-Bit breiten Kontrollsignal „ALUOp“ ist der Wert der 3 Signalleitungen bei der ALU beschriftet.

Betrachten wir zuerst den Befehl: „Jump and Link“: Opcode ist 7: 000111. Die Bits 5-0, die bei einem R-Format das Feld „funct“ beinhalten, können wir beliebig annehmen, denn da das Most Significant Bit des Opcodes gleich 0 ist, haben diese Bits keinen Einfluss in der Control Logic. Die ControlLogic ist in Abb. 5.17 gezeigt. In Abb. 5.18 ist die Rechnerstruktur gezeigt, wobei die durchlaufenen Datenpfade und die auf 1 gesetzten Kontrollsignale hervorgehoben sind.

Als zweites Beispiel ist die ControlLogic und das Rechnerschaltbild für den Befehl „set on less than“ gezeigt. Der Opcode zu diesem R-Format Befehl ist 32: 100000. Hier wird also auch das Feld „funct“ aus den Instruktionsbits 5-0 benötigt, um die ALU-Instruktion genau zu bestimmen. Das „funct“-Feld trägt den Wert 7:000111. Die Control Logic ist in Abb. 5.19 gezeigt. Die Abb. 5.20 zeigt wieder die Auswirkung auf die gesamte Rechnerstruktur.

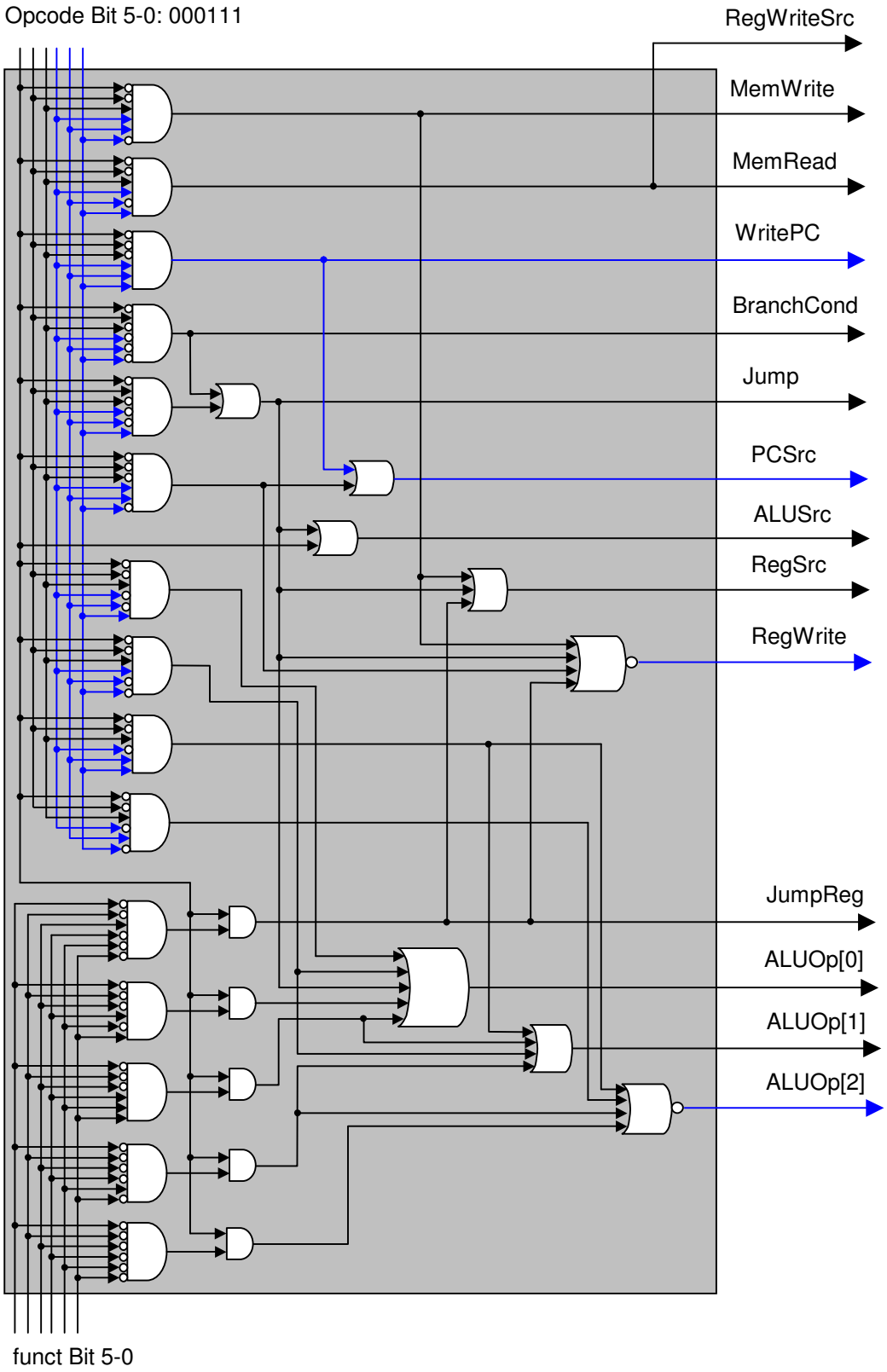


Abb. 5.17: Die Control Logic mit den hervorgehobenen 1 Signalen bei der Auswertung des Opcodes 7: 000111.

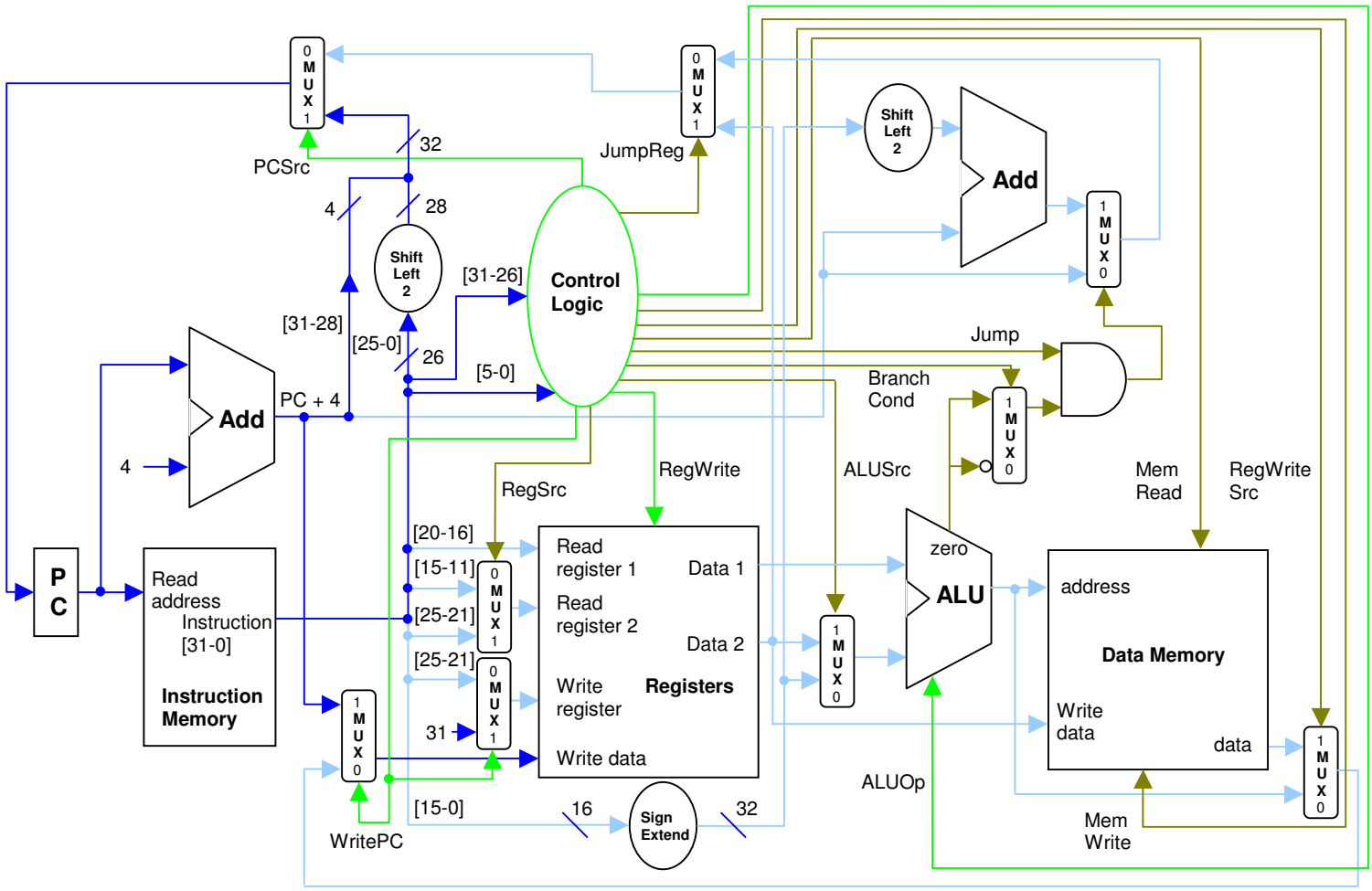


Abb. 5.18: Die komplette Rechnerstrukur bei der Auswertung einer „Jump and Link“ Instruktion. Die aktiven Datenleitungen sind blau hervorgehoben, die Kontrollsignale mit einer 1 sind hellgrün hervorgehoben.

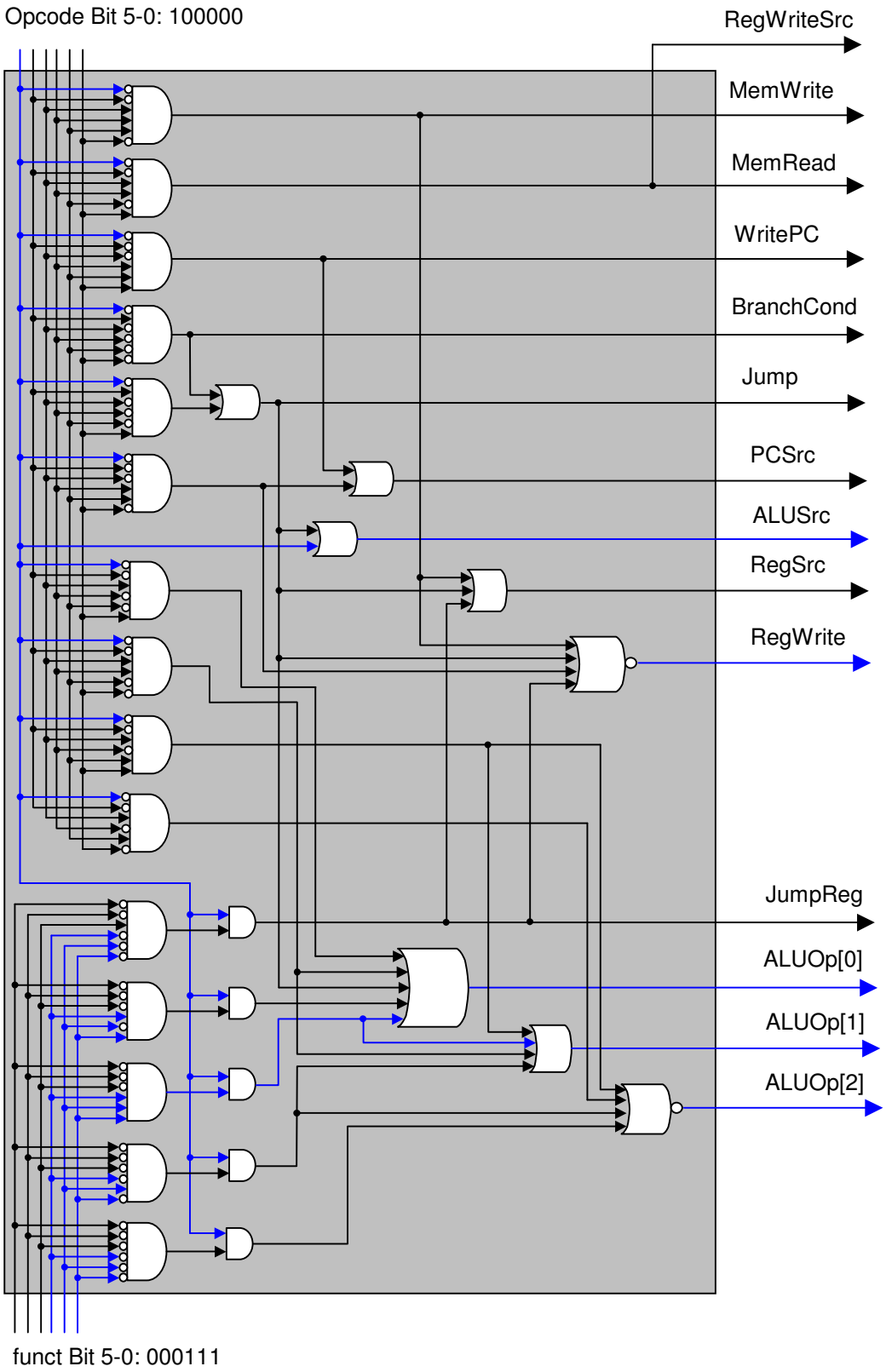


Abb. 5.19: Die Control Logic mit den hervorgehobenen 1 Signalen bei der Auswertung des Opcodes 32: 100000. Dieser Opcode verweist auf eine R-Format Instruktion. Das Feld „funct“ aus den Instruktionsbits 5-0 wird durch das Opcode-Bit [5] mit einbezogen.

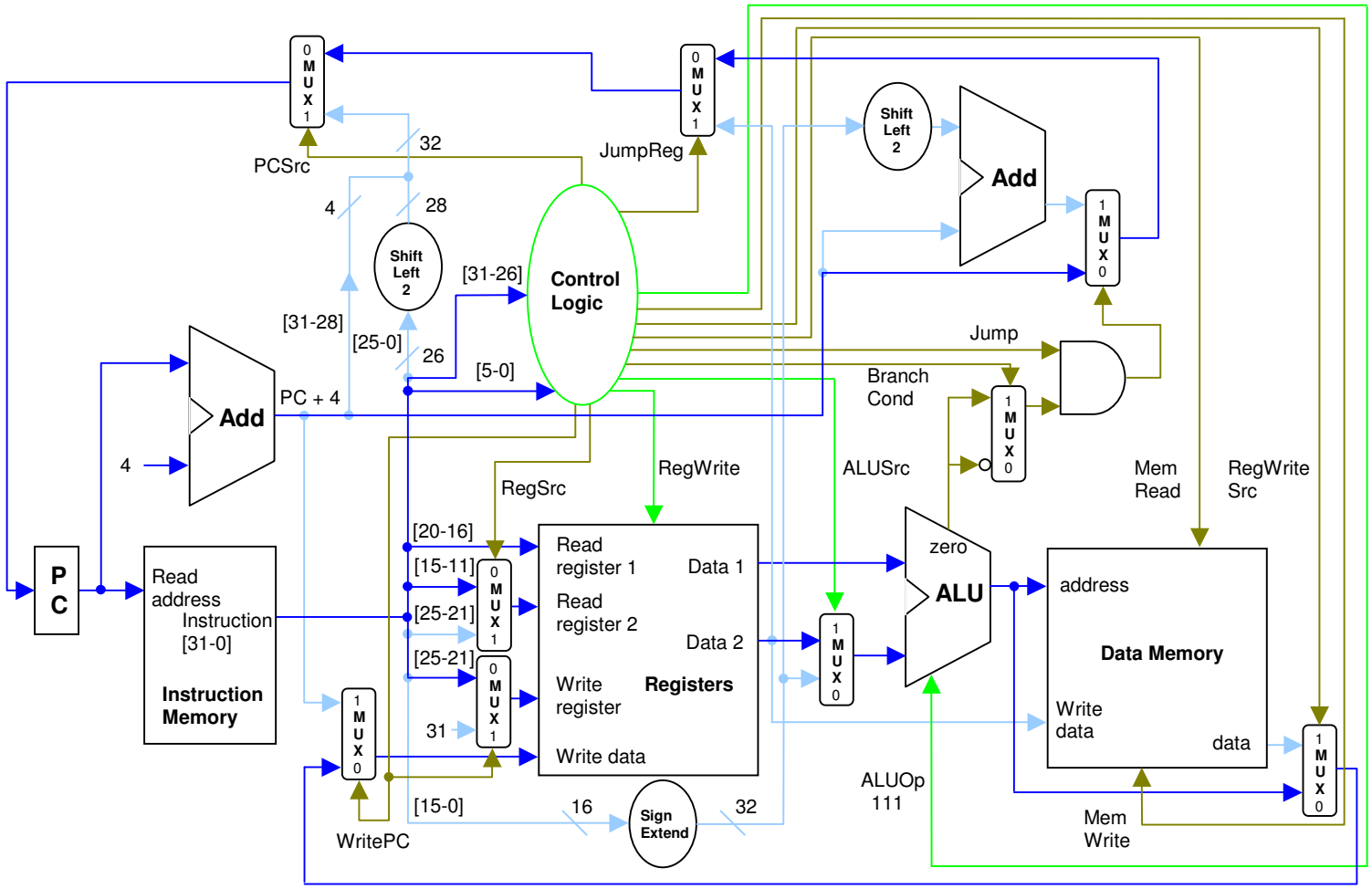


Abb. 5.20: Die komplette Rechnerstrukur bei der Auswertung einer „Set on less than“ Instruktion. Die aktiven Datenleitungen sind blau hervorgehoben, die Kontrollsignale mit einer 1 sind hellgrün hervorgehoben.

Aufgabe 5.5: Zeichnen Sie die gesetzten Kontrollsignale gemäss ihren Ergebnissen aus Aufgabe 5.4 in die Rechnerschaltbilder, die Sie in Aufgabe 4.1 erstellt haben. Überprüfen Sie dabei, ob die Komponenten für die benötigten Datenpfade richtig gesteuert werden.

Zusammenfassung

Wir haben in diesem Kapitel zuerst gesehen, wie man die benötigten Kontrollsignale für eine bestimmte Instruktion zusammenträgt. Dies haben wir dann für jede Instruktion in unserem Instruction Set getan und die Ergebnisse in einer Tabelle zusammengetragen. Aus der Tabelle konnte dann ausgelesen werden, für welche Instruktions-Opcodes ein Signal zu einer 1 oder einer 0 auswertet.

Im zweiten Abschnitt wurde dann für jedes Kontrollsignal eine Schaltung vorgestellt, die bei den entsprechenden Opcodes das richtige Signal liefert. All diesen Kontrollsignal-Schaltungen haben wir dann zu einer grossen Komponente zusammengefasst: Die Control Logic.

Diese Control Logic haben wir in das Rechnerschaltbild eingebaut, indem die Opcode-Bits und die „funct“-Feld Bits einer Instruktion in die Control Logic geleitet wurden. Die Ausgangssignale mussten dann nur noch in die entsprechenden Komponenten geleitet werden. Durch diese Control Logic werden nun die Opcodes der Instruktionen ausgewertet, und die Komponenten je nach Instruktion gesteuert.

Zum Abschluss wurden an zwei Beispielen aufgezeigt, wie in der Control Logic die benötigten Kontrollsignale aus einem Opcode ermittelt, und dadurch der Datenfluss durch die Komponenten gemäss den Instruktionen gesteuert wird.

Lernziele

Haben Sie die folgenden Lernziele des Kapitels erfüllt? Falls Sie sich nicht ganz sicher fühlen, schlagen Sie die Punkte nochmals nach. Fühlen Sie sich beim Umgang mit dem Stoff sicher? Dann können Sie den Kapiteltest in Angriff nehmen.

- Sie wissen, was die Aufgabe einer Control Logic im Rechner ist
- Sie wissen, wie die benötigten Kontrollsignale für die Ausführung einer bestimmten Instruktion identifiziert werden können
- Sie können eine Schaltung für ein bestimmtes Kontrollsignal entwerfen
- Sie können für eine gegebene Instruktion die Werte im Schaltbild der Control Logic einzeichnen.

Kapiteltest 5

Aufgabe 1

Wie könnte die Kontrollsignal-Schaltung „RegWrite“ benutzt werden, um die Signale für die Kontrolllleitung „RegSrc“ zu erhalten? Welche Werte müssen dabei für die x-en in der Spalte für das Kontrollsignal „RegSrc“ angenommen werden?

Aufgabe 2

Zeichnen Sie auf den Controll Logic-Vorlagen die benötigten Pfade ein, die bei den folgenden Instruktionen durchlaufen werden.

a) „Jump and Link“
b) „StoreWord“
c) „Jump-Register“
d) “Sub”

Vergessen Sie nicht auf ihrer Schaltbild-Vorlage die Aufgaben-Nummer und ihren Namen anzugeben.

Aufgabe 3

Zeichnen Sie auf den Schaltbild-Vorlagen die benötigten Pfade ein, die bei den Instruktionen aus Aufgabe 2 durchlaufen werden. Vergessen Sie nicht auf ihrer Schaltbild-Vorlage die Aufgaben-Nummer und ihren Namen anzugeben.

Bonusaufgabe

Wie könnten die Schaltungen für die Kontrollsignale „Jump“, „JumpReg“, „ALUSrc“, oder „PCSrc“ optimiert werden? Wählen Sie eine der Schaltungen aus, und zeichnen Sie ihre optimierte Schaltungsversion. Begründen Sie mit einem kurzen Text, warum Ihre Optimierungen angebracht sind.

Kapitel 6: Ein Rückblick

Einleitung

Das Design unseres Rechners ist nun vollständig erarbeitet worden. In diesem Kapitel wollen wir uns zum Schluss nochmals einen Überblick über die ganze Rechnerstruktur verschaffen, und uns mit den Vor- und Nachteilen dieser Rechnerstruktur auseinander setzen. Hier gäbe es noch sehr viel Spielraum für Optimierungen, die wir aber nur sehr oberflächlich ansprechen können. Sicherlich werden Sie auch mit einigen der gemachten Annahmen nicht einverstanden sein, da sie für einen modernen Rechner unrealistisch erscheinen. Und wo bleibt der Bildschirm, die Tastatur und die Maus? Auch hierzu kann nur sehr knapp darauf eingegangen werden, da man mit dieser Materie problemlos Regale von Büchern füllen könnte. Mit einigen Ausführungen dazu soll aber dem Leser ein abgerundetes Bild eines Computers vermittelt werden. Das Kapitel ist in 3 Abschnitte unterteilt:

1 In ersten Abschnitt wird eine Zusammenfassung der Schritte zu unserem Rechnerdesign gegeben. Hier werden auch kurz die Vor- und Nachteile unserer Rechnerstruktur angesprochen.

2 Im zweiten Abschnitt werden die im Leitprogramm getroffenen Annahmen etwas ausgeleuchtet, und es wird erläutert, was denn nun passiert, falls bei einem Computer auf den Einschaltknopf gedrückt wird.

3 In Abschnitt 3 wird sehr vereinfacht erklärt, wie nun die Tastatur oder eine Harddisk im Rechner eingebaut werden könnte und welche Zwecke ein Betriebsystem erfüllt. Ziel dieses Abschnitts wird sein, dem Leser klar zu machen, dass ein Computer nichts mit einem Wunder zu tun hat, sondern dass sein Zauber von den extremen Komplexitäten und ihrem Zusammenspiel ausgeht.

6.1 Zusammenfassung und Analyse

Im ersten Kapitel wurden die für den Rechner benötigten Hardware-Komponenten vorgestellt. Aus sehr grundlegenden Bausteinen wie Drahtleiter, AND- OR- und NOT Gatter sowie einfachen Speicherbausteinen konnten wir uns die nötigen Komponenten wie die ALU, den Register-Block und Memory-Blöcke zusammentragen. Die Geschichte hat gezeigt, dass ein solcher Entwurf von Hardware-Komponenten keinesfalls etwas Triviales ist, hat doch erst der Aufbau der logischen Gatter aus der schnellen Halbleitertechnologie und deren Anordnung auf kleinstem Raum auf einem „Silicium-Waver“ zur explosionsartigen Entwicklung der höheren Rechnerstrukturen geführt.

Im zweiten Kapitel wurde ein noch abstraktes, aber sehr gängiges Prinzip vorgestellt, wie die vom Menschen verständlichen und von ihm entwickelten Programmbefehle und Programmabläufe auf ein sehr kleines aber mächtiges Set von Anweisungen reduziert werden kann. D.h. es sollte zum Ausdruck gebracht werden, wie die Anweisungen von einer vom Menschen gut verständlichen Programmiersprache wie z.B. „C“ in Assembler übersetzt werden kann, und diese Assemblerprogrammiersprache weiter auf ein Instruction Set für einen spezifischen Rechner reduziert wird.

Das dritte Kapitel hat eine Möglichkeit aufgezeigt, wie die Befehle des Instruction Set in einen 32-Bit Maschinencode übersetzt werden können. Dabei beinhaltete der Maschinencode einerseits die benötigten Parameter zu einem Befehl, andererseits einen vordefinierten Teil, der die Instruktion für den Rechner eindeutig bestimmen lässt: den Opcode. Der Maschinencode bildet also die eigentliche Schnittstelle, die den Menschen mit der Maschine kommunizieren lassen kann.

Mit all diesem Vorwissen konnte nun eine Schaltung entworfen werden, die einerseits dafür sorgt, dass bei jedem Clock-Schlag eine nächste Instruktion-Adresse vorbereitet wird, andererseits die Anweisungen der momentanen Instruktion über die Hardware-Komponenten wie ALU und Memory umgesetzt werden. Im Kapitel 4 wurde die Schaltung nur für die Datenpfade, also die Befehlsparameter einer Instruktion, betrachtet und ausgebaut. In Kapitel 5 wurde dann die Rechnerstruktur für die Kontrollpfade, also die Steuerung der Hardware-Komponenten über den Opcode-Teil der Instruktion, vervollständigt.

Als Ergebnis haben wir einen Rechner erhalten, der für vielseitige Anwendungszwecke und Berechnungen programmiert werden könnte. Dabei wird im Rechner bei jedem Clock-Schlag eine einzelne Maschinencode-Instruktion abgearbeitet. Eine solche Rechnerstruktur wird aus diesem Grund „Single-Cycle“ Rechnerstruktur („Einzelner Zyklus“) genannt, die mit unseren Instruktionsformaten und Opcodes in einem MIPS-Rechner auch tatsächlich umgesetzt wurde.

Die Dauer eines einzelnen Clock-Zyklus ist dabei für die Geschwindigkeit des Rechners von entscheidender Bedeutung. Ein Clock-Zyklus muss dabei genau so lange dauern, dass alle Hardware-Komponenten bei einer Instruktion die Signale entsprechend schalten und weiterleiten können. Die meiste Zeit eines Clock-Zyklus wird für den Memory Zugriff benötigt. Da Zugriffe ins Daten Memory im Verhältnis zum Register-Block oder der ALU sehr lange dauern, und für jeden Zyklus diese Zugriffszeit angenommen werden muss, auch wenn das Daten Memory gar nicht für die Instruktionsanweisung benutzt wird, wird dadurch unsere Rechnerstruktur mächtig ausgebremst. Eine sehr verbreitete Optimierung dazu bilden die sogenannten Cache-Speicher. Dies sind spezialisierte Speicherbausteine, die einen schnelleren Datenzugriff erlauben. In ihnen werden die momentan am häufigsten benutzten Daten gespeichert um den Zugriff zu beschleunigen. „Multi-Cycle“ Rechnerstrukturen und „Pipelined“-Rechnerstrukturen widmen sich auch genau dieser Problematik, und sorgen dafür, dass alle Rechnerkomponenten bei jedem Zyklus optimal ausgelastet werden. An dieser Stelle muss auf entsprechende Fachliteratur verwiesen werden, dazu aber nur noch so viel: Das Schaltbild für einen „Pipelined“-Rechner unterscheidet sich von unserem „Single-Cycle“ Rechner grundsätzlich nur durch 4 zusätzlich eingebaute einzelne Register!

Unsere „Single-Cycle“ Rechnerstruktur hat aber auch einen nicht zu vernachlässigenden Vorteil: da für jede Instruktion ein einzelner Clock-Zyklus von fixer Dauer aufgewendet wird, lässt sich exakt vorherbestimmen, wie lange der Rechner für die Abarbeitung eines Programms braucht. Dies ist bei „Multi-Cycle“ und „Pipelined“-Rechnerstrukturen nicht mehr der Fall.

6.2 Register-Window und das Starten eines Rechners

Zuerst soll an dieser Stelle noch eine Antwort gegeben werden, warum im Kapitel 2 beim Stack-Management die Register $s0-$s7 auf dem Stack mitgespeichert werden:

Es handelt sich hier um eine Optimierung auf Hardware und Software-Seite in Anlehnung an die Sparc-Rechnerarchitektur. Um möglichst die vielen Daten Memory-Zugriffe beim Prozedurwechsel zu vermeiden, wird einfach mal angenommen, dass Prozedurwechsel eher selten vorkommen. Dann könnten die Daten in den Registern belassen werden, und bei einer neuen Prozedur wird einfach ein neuer Satz von Registern verwendet. Der Sparc-Rechner hat insgesamt 128 Register, und einer Prozedur werden wie gehabt 32 Register zugewiesen. Werden aber mehr als 4 verschachtelte Prozeduraufrufe gemacht, wie dies bei rekursiven Prozeduren oft der Fall ist, müssen die Daten in den Registern nachträglich im Stack gespeichert werden, und dabei wird der frei gehaltene Platz im Stack benötigt. Diese Technik wird „Register-Window“ genannt, und sollte Ihnen zeigen wie vielfältig die Ansätze für Optimierungen sind.

Nun zum Starten eines Rechners: Haben Sie die Annahme aus Kapitel 1 gerechtfertigt empfunden, dass das Programm im Instruction Memory bereits vorhanden ist? Dies ist selbstverständlich nicht einfach so der Fall, es wurden aber tatsächlich zu Beginn der Rechnerentwicklung Lochkarten und etwas später Magnetbänder dazu verwendet, den Programmcode in das Instruction Memory zu laden um diesen dann abzuarbeiten. Vielleicht erinnern Sie sich noch, dass vor nicht all zu langer Zeit eine Diskette benutzt werden musste, um die Programme zu laden. Heutzutage funktioniert es grundsätzlich nicht gross anders.

Wird bei einem Rechner auf den Einschalt-Knopf gedrückt, wird eine fest eingebaute elektronische Schaltung aktiv die sicherstellt, dass alle Hardware-Komponenten genügend mit Strom versorgt sind. Danach wird ein „Reset“ Signal an den Prozessor ausgegeben. (für das „Reset“ Signal gibt es oft einen Knopf am Rechnergehäuse). Im Prozessor werden dadurch fest vorbestimmte Werte in die Register geladen, und die Clock-Taktung wird gestartet. Der Prozessor wird erst dann aktiv und beginnt mit der Abarbeitung eines ganz speziellen Programms, das in einem spezialisierten Memory-Baustein (ROM: Read Only Memory) fix vorhanden ist, dem sogenannten BIOS (Basic Input Output System). Das BIOS entspricht also unserer Lochkarte für das Instruction Memory, wie sie im

Leitprogramm angenommen wurde. Dazu muss natürlich eine funktionierende Clock vorhanden sein, die grundsätzlich wie eine Quarzuhr aufgebaut ist.

Dieser BIOS-Code aus dem ROM-Baustein überprüft zum Beispiel, was für eine ALU und wie viel Speicher im Rechner sind und welche sonstigen Geräte wie Harddisks, Laufwerke, Tastatur, Soundkarte etc. angeschlossen wurden. Sie können sich das BIOS und seine gesammelten Daten beim Rechnerstart normalerweise durch eine „F“ Taste oder die „Esc“ Taste anschauen und auch anpassen.

Im BIOS wird auch angegeben, wo der Rechner nach dem Starten nach einem ersten auszuführenden Programm suchen soll. Dies kann zuerst die Diskette, dann ein CD-Rom Laufwerk und danach die Festplatte sein (Boot-Reihenfolge). Normalerweise wird dann ein Programm in einem fix vordefinierten Bereich auf der Festplatte gestartet, der sogenannte „Boot-Manager“.

Der Boot-Manager ist wiederum ein kleines Programm, dass bestimmt, welches Programm als nächstes gestartet werden soll. Dies ist dann normalerweise ein Betriebsystem, ein Programm das läuft bis der Rechner wieder ausgeschaltet wird. Sind mehrere Betriebsysteme auf der Festplatte vorhanden, so kann der Boot-Manager dem Benutzer ein Auswahl-Menü bieten, welches der Betriebsysteme gestartet werden soll. Was ein Betriebsystem so alles beinhaltet werden Sie ansatzweise im nächsten Abschnitt erfahren.

6.3 Geräte, OS und Optimierungen

Wir haben bereits gesehen, wie das BIOS kontrolliert, welche externen Geräte wie Festplatte, Modem, Bildschirm, Scanner u.s.w. an den Rechner angeschlossen wurden. Wie kommuniziert aber der Rechner mit diesen Geräten?

Diese Frage müsste im Detail von Gerät zu Gerät unterschiedlich beantwortet werden, was aber allen Geräten gemeinsam ist, ist dass sie vom Hersteller ein sogenannte „Schnittstelle“ (Interface) definiert bekommen. D.h. der Geräte-Hersteller gibt an, welche Befehle (binäre Signale) an das Gerät gesendet werden müssen, um die gewünschten Ergebnisse (wieder binäre Signale) zu bekommen. Das Gerät wird dann bei einer vorgesehenen Hardware-Schnittstelle im Rechner eingebaut oder angeschlossen. Sehr vereinfacht kann man sich dies nun so vorstellen:

Jede Hardware-Schnittstelle ist wie eine oder zwei dem Gerät zugewiesene „virtuelle“ Memory-Adresse., je nach dem ob das Gerät nur einen Input benötigt, z.B. der Bildschirm oder ein Drucker, ob das Gerät nur einen Output hat, z.B. Tastatur oder Maus, oder ob das Gerät einen Input und einen Output benötigt, wie dies bei einem Modem der Fall wäre. Diese zugewiesenen „virtuellen“ Memory-Adressen sind mit 0 initialisiert.

Möchte man nun einen Befehl z.B. an die Festplatte senden, wird mit einem normalen sw-Assemblerbefehl in die dem Gerät zugewiesene Input- Memory-Adresse geschrieben. Danach müssen einige Takt-Zyklen gewartet werden, bis das Gerät ein Resultat berechnet hat. Dies ist dann z.B. daran erkennbar, das die Input-Memory-Adresse durch das Gerät wieder auf 0 gesetzt wurde, also das Gerät wieder einen neuen Befehl empfangen kann, oder daran, das die Output-Memory-Adresse nicht mehr 0 ist, also ein Resultat darin abgelegt wurde. Wenn das Resultat durch den Rechner empfangen wurde, muss dieser die Output-Memory-Adresse wieder auf 0 setzten, um dem Gerät anzuzeigen, das ein nächstes Resultat empfangen werden kann.

Man kann sich nun einfach vorstellen, welche Herausforderung es ist, den Überblick über die verschiedenen Geräte-Befehle, die Adresszuweisungen und deren Input/Output-Signale zu bewahren. Hinzu kommt, dass die Daten auf z.B. einer Festplatte organisiert werden wollen, dass den Geräten Prioritäten zugewiesen werden müssen und dabei einem Benutzer noch die Möglichkeit gegeben werden soll, nach seinen Wünschen bestimmte Aktionen mit den Geräten auszuführen. All diese Aufgaben werden von einem Betriebsystem (Operating System, kurz OS) übernommen, das einem Benutzer eine intuitive Abstraktionsebene für den Umgang und die Organisation seiner Geräte liefert.

Das Betriebsystem kann aber noch vieles mehr. Z.B. ist eine weitere wichtige Aufgabe des Betriebsystems den Computer vor dem Benutzer selbst zu schützen, denn was soll passieren, falls bei einer Rechenaddition in einem Programm ein Overflow passiert, oder der Stack so gross geworden

ist, dass er im Daten Memory keinen Platz mehr findet, oder unglücklicherweise eine Rechendivision durch 0 im Instruktionscode verlangt wird? Hier kann das Betriebsystem durch Hardware-Signale und Fehlerroutinen in der Software die Fehlerquelle herausfinden und kann sich dann durch eine bestimmte Aktion selber weiterhelfen, auch wenn es den Abbruch des ausgeführten Programms bedeutet. Wenigstens kann dann über das Betriebsystem weiter gearbeitet werden.

Es sollte damit zum Schluss deutlich gemacht worden sein, wie viele verschiedene Details, Optimierungen auf Hardware und Software-Seite, wie viele verstrickte Fehlerquellen zu beachten und umzusetzen sind und wie viele tausend Zeilen Code fehlerlos geschrieben werden müssen, um einen Computer vor uns zu haben, wie wir es uns gewohnt sind. Ausserordentlich faszinierend, für mich keine Frage...

Zusammenfassung

Im ersten Abschnitt dieses Kapitels wurden nochmals die einzelnen Schritte angesprochen, die zum Design des programmierbaren Rechners führten. Dabei wurde der grosser Nachteil der langen Clock-Zyklen aufgrund des Daten-Memorys ermittelt. Als Vorteil haben wir die Berechenbarkeit der benötigten Zeitdauer für ein Programm genannt. Im zweiten Abschnitt haben wir zuerst ein Beispiel für die vielfältigen Optimierungsmöglichkeiten kurz vorgestellt, und danach erörtert, was beim Einschalten eines modernen Computers zuerst alles passiert, bevor ein eigenes Programm abgearbeitet werden kann. Der dritte Abschnitt erklärte, wie man sich die Kommunikation mit Geräten, z.B. Bildschirm und Tastatur einfach vorstellen kann. Für die schnell zunehmende Komplexität innerhalb des Computers wurde ein Betriebsystem eingeführt, das die Schnittstelle Mensch zu Maschine zu abstrahieren vermag.

Lernziele

Dieses Kapitel hat keinen Kapiteltest zum Abschluss, da es sich um sehr abstrakte Erläuterungen handelte, die als Additium das Bild eines Computers abrunden soll. Trotzdem kann man auch aus diesem Kapitel was lernen.

- Sie kennen einen Vorteil und einen Nachteil der entwickelten Rechnerstruktur. Was für Ansätze fallen ihnen spontan ein, dieses Problem zu beheben?

- Sie können mit eigenen Worten beschreiben, was beim Starten eines Rechners alles passiert. Wo ist die „Lochkarte“ in einem modernen Rechner zu finden?

- Sie können mit einer eigenen Grafik aufzeigen, wie die Kommunikation mit einem externen Gerät funktionieren könnte

- Sie wissen, welche Funktionen ein Betriebsystem übernimmt.

Anhang A

A.1: Die Registerkonvention

Name	Index	Gebrauch	Auf Stack
$zero	0	Beinhaltet immer 0	Nein
$at	1	Vom Linker verwendet	Nein
$v0-$v1	2-3	Prozedur-Resultate	Nein
$a0-$a3	4-7	Prozedur-Argumente	Nein
$t0-$t7	8-15	Für temporäre Zwischenwerte	Nein
$s0-$s7	16-23	Prozedurvariablen	Ja
$Hi-$Lo	24-25	unterstützt Multiplikation und Division	Nein
$k0-$k1	26-27	Vom Betriebsystem verwendet	Nein
$gp	28	Pointer auf globale Variablen	Ja
$sp	29	Stackpointer	Ja
$fp	30	Framepointer	Ja
$ra	31	Prozedur-Rücksprungadresse	Ja

A.2 Das Instruction Set

Operation	Assembler-Befehl	Wirkung	Format
Addition	add $c, $a, $b	$c = $a + $b	R
Subtraktion	sub $c, $a, $b	$c = $a – $b	R
Logisches AND	and $c, $a, $b	$c = $a & $b	R
Logisches OR	or $c,$a, $b	$c = $a | $b	R
Set on Less than	slt $c, $a, $b	if $a < $b then $c = 1 else $c = 0	R
Addition Im.	addi $c, $a, K	$c = $a + K	I
Subtraktion Im.	subi $c, $a, K	$c = $a – K	I
Logisches AND Im.	andi $c, $a, K	$c = $a & K	I
Logisches OR Im.	ori $c,$a, K	$c = $a | K	I
Set on Less than Im.	slti $c, $a, K	if $a < K then $c = 1 else $c = 0	I
Load Word	lw $c, K($a)	$c = Mem[K + $a]	I
Store Word	sw $c, K($a)	Mem[K + $a] = $c	I
Branch on equal	beq $a, $b, K	if $a == $b then PC = K	I
Branch on not equal	bne $a, $b, K	if $a <> $b then PC = K	I
Jump	j K	PC = K	J
Jump and Link	jal K	$ra = PC + 4, PC = K	J
Jump Register	jr $ra	PC = $ra	R

A.3: Instruktionsformate

Instruktionsformat der Klasse J-Format

Bit 31 -26	Bit 25 - 0
Opcode	Sprung-Adresse

Instruktionsformat der Klasse I-Format

Bit 31 -26	Bit 25 - 21	Bit 20 - 16	Bit 15 - 0
Opcode	Register A	Register B	Konstante

Instruktionsformat der Klasse R-Format

Bit 31 -26	Bit 25 - 21	Bit 20 - 16	Bit 15 - 11	Bit 10 - 6	Bit 5 - 0
Opcode	Register A	Register B	Register C	shamt	funct

A.4: Opcodes für das Instruction Set

Operation	Befehl	Format	Opcode	funct
Addition	add	R	32	4
Subtraktion	sub	R	32	5
Logisch AND	and	R	32	0
Logisch OR	or	R	32	2
Set on less than	slt	R	32	7
Jump Register	jr	R	32	8
Addition Im.	addi	I	8	-
Subtraktion Im.	subi	I	9	-
Logisch AND Im.	andi	I	10	-
Logisch OR Im.	ori	I	11	-
Set on less than Im.	slti	I	12	-
Load Word	lw	I	13	-
Store Word	sw	I	14	-
Branch on equal	beq	I	16	-
Branch on not equal	bne	I	17	-
Jump	j	J	6	-
Jump and Link	jal	J	7	-

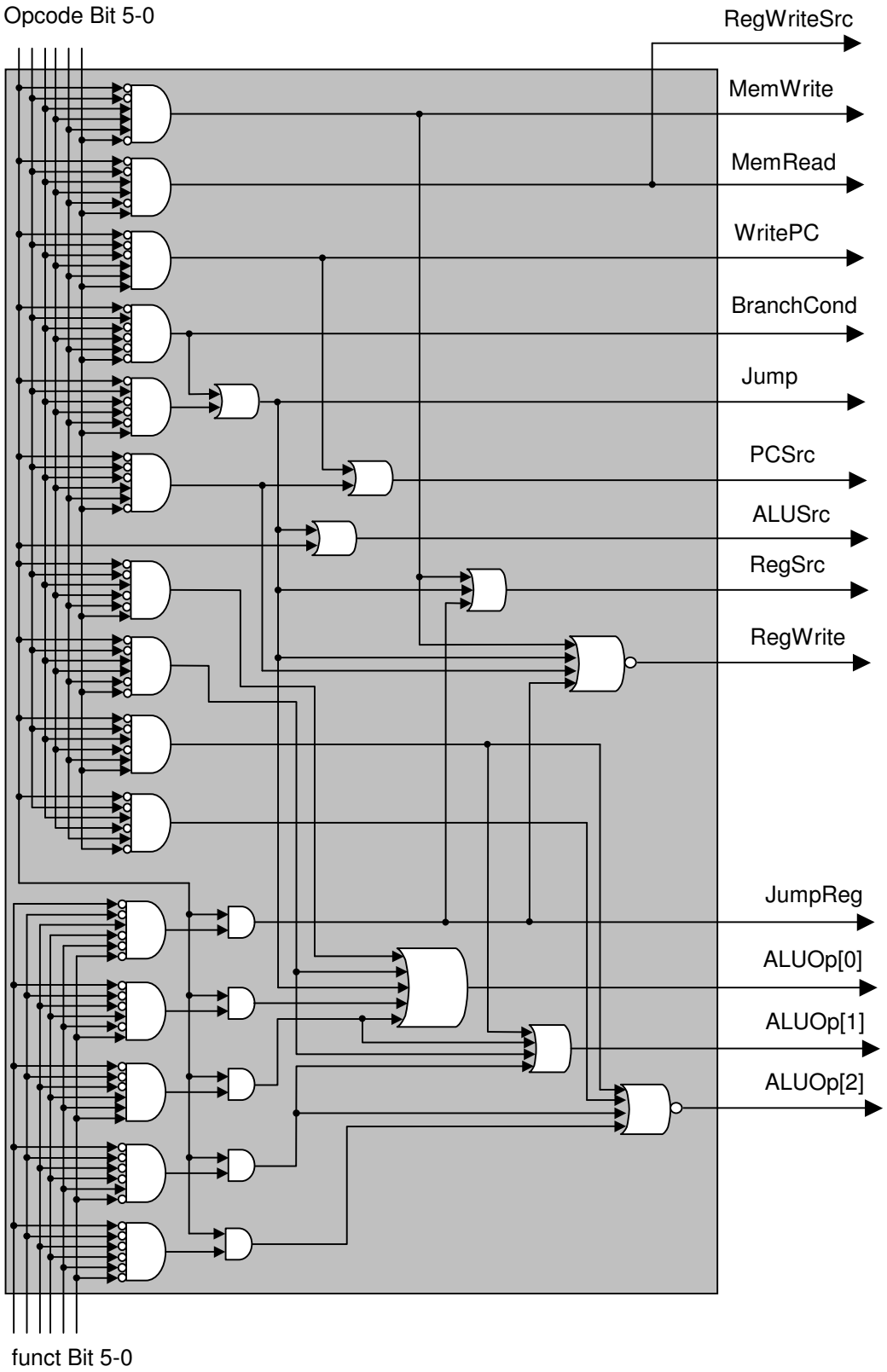
A.5: Die Kontrollsignale der Rechnerstruktur

Kontrollsignal	Funktion
RegWrite	Es wird ein Wert in ein Register geschrieben
ALUSrc	Leitet als zweiten Operanden entweder einen Registerwert oder die Instruktionskonstante aus Bit 15-0 zur ALU
MemRead	Es wird ein Wert aus dem Data Memory gelesen
MemWrite	Es wird ein Wert in das Data Memory geschrieben
RegWriteSrc	Es wird entweder das Resultat aus der ALU oder ein Wert aus dem Data Memory zum Register-Block zurückgeleitet.
WritePC	Falls auf 1, wird der PC+4 Wert am „Write data“ Eingang, und die Registeradresse 31 am „Write register“ Eingang des Register-Blocks angelegt.
RegSrc	Speist die Registeradresse in Bit 25-21 oder die Registeradresse in Bit 15-11 in den „Read register 2“ Eingang des Register-Blocks ein
ALUOp	Operationscode der ALU bestehend aus dem Kontrollsignal Subtraction und dem 2bit breiten Operationssignal. Siehe Abb. 1.21
BranchCond	Unterscheidet „Branch on equal“ und „Branch on not equal“
Jump	Signalisiert einen unbedingten Sprung
JumpReg	Leitet die Sprungadresse des Befehls „Jump Register“ zum PC-Register
PCSrc	Leitet die Sprungadresse des Befehls „Jump“ oder des Befehls „Jump and Link“ zum PC-Register

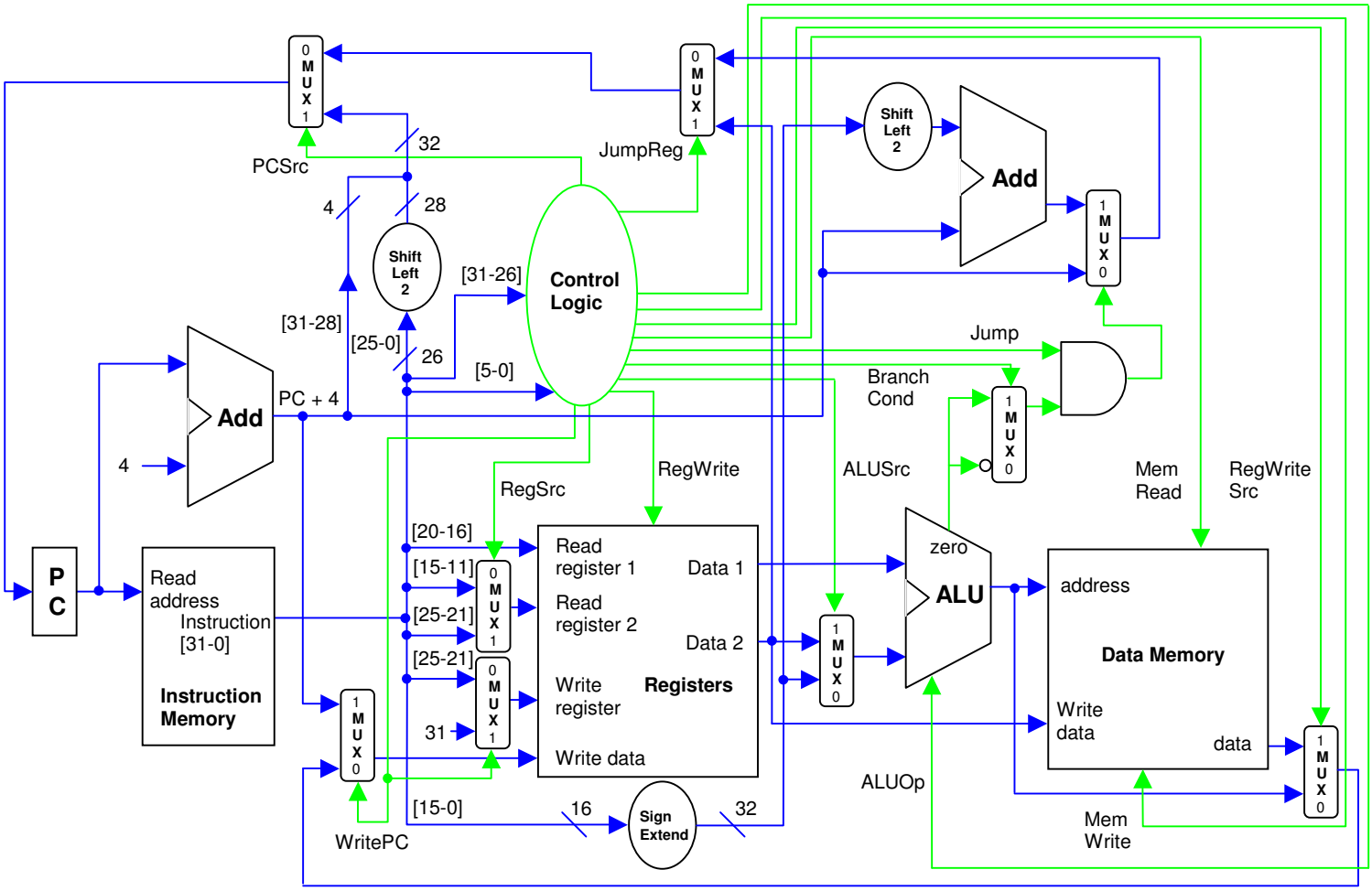
A.6: Die Kontrollsignale und Opcodes des Instruction Set

Befehl	Opcode	funct	Reg Write	ALU Src	Mem Read	Mem Write	RegWrite Src	Write PC	Reg Src	ALU Op	Branch Cond	Jump	JumpReg	PCSrc
add	32	4	1	1	0	0	0	0	0	100	x	0	0	0
sub	32	5	1	1	0	0	0	0	0	101	x	0	0	0
and	32	0	1	1	0	0	0	0	0	000	x	0	0	0
or	32	2	1	1	0	0	0	0	0	010	x	0	0	0
slt	32	7	1	1	0	0	0	0	0	111	x	0	0	0
addi	8	-	1	0	0	0	0	0	x	100	x	0	0	0
subi	9	-	1	0	0	0	0	0	x	101	x	0	0	0
andi	10	-	1	0	0	0	0	0	x	000	x	0	0	0
ori	11	-	1	0	0	0	0	0	x	010	x	0	0	0
slti	12	-	1	0	0	0	0	0	x	111	x	0	0	0
lw	13	-	1	0	1	0	1	0	x	100	x	0	0	0
sw	14	-	0	0	0	1	x	0	1	100	x	0	0	0
beq	16	-	0	1	0	0	x	0	1	101	1	1	0	0
bne	17	-	0	1	0	0	x	0	1	101	0	1	0	0
j	6	-	0	x	0	0	x	0	x	x	x	0	0	1
jal	7	-	1	x	0	0	x	1	x	x	x	0	0	1
jr	32	8	0	x	0	0	x	0	1	x	x	0	1	0

A.7: Die Control Logic



A.8: Das Rechnerschaltbild

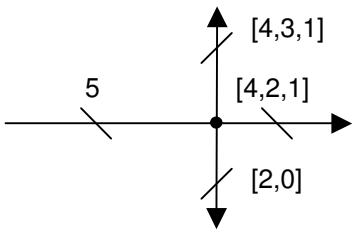


Anhang B

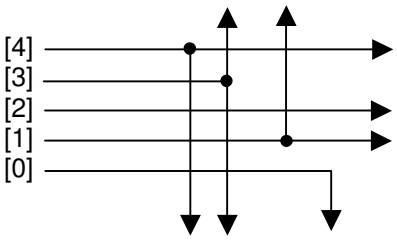
Musterlösungen:

Aufgabe 1.1

a)



b)



Aufgabe 1.2

Aufgabe 1.3

a) Es wurde angenommen, dass die 2 vorne abgeschnittenen Bits 0 sind. Sind diese beiden Bits nicht 0 so ist das Multiplikationsresultat falsch, wie man anhand eines Beispiels sehen kann: Wir lassen den Wert 24 auf einer 5 Bit breiten Leitung durch eine Shift left 2 –Komponente fliessen

   Der dezimale Wert 24 entspricht in einer 5 Bit breiten binären Darstellung 11000.
   Shift left 2 auf den Wert 11000 ergibt 00000, also in Dezimal den Wert 0.
   4 * 24 würde aber 96 ergeben. (in binärer Darstellung 1100000)

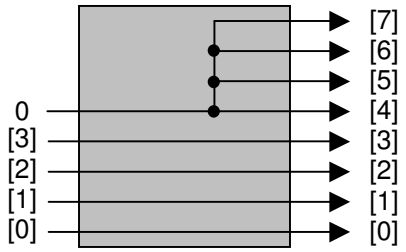
   ⇨ Würde man den Wert 24 in eine 7 Bit breite Leitung einspeisen (0011000) und dann shiften (1100000), wäre das Ergebnis korrekt.

b) Ein Shift left 4 würde einer Multiplikation mit 16 entsprechen, auch wieder unter der Annahme, dass die 4 abgeschnittenen Bits den Wert 0 tragen. Als Beispiel lassen wir den Wert 2 auf einer 6 Bit breiten Leitung durch eine Shift left 4 –Komponente fliessen

   Der dezimale Wert 2 entspricht in einer 6 Bit breiten binären Darstellung 000010.
   Shift left 4 auf den Wert 000010 ergibt 100000, also in Dezimal den Wert 32.

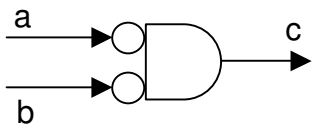
Aufgabe 1.4

Die neu hinzukommenden Leitungen würden immer den Konstanten Wert 0 tragen. Dadurch werden nur die Leitungen erweitert, am Wert auf den Leitungen ändert sich nichts.



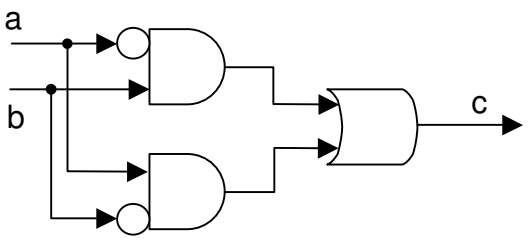
Aufgabe 1.5

a)



a	b	c
0	0	1
0	1	0
1	0	0
1	1	0

b)



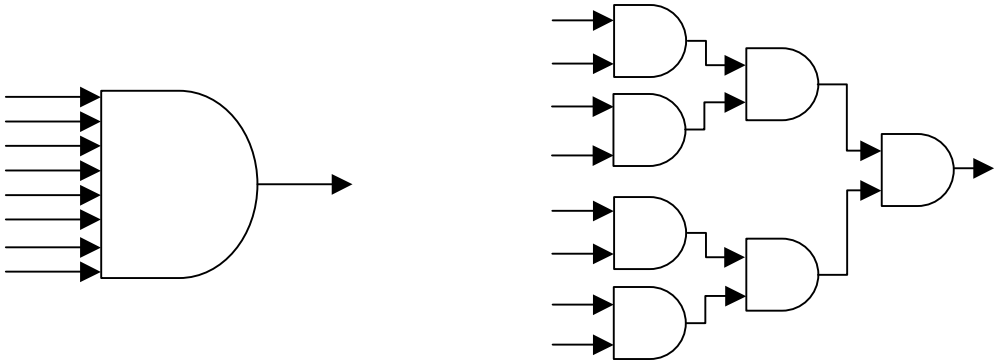
c)

a	b	c	d
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Aufgabe 1.6

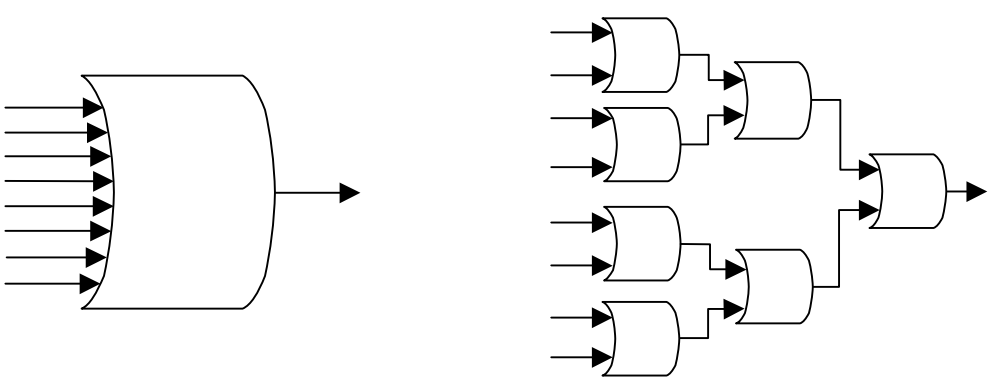
a)

b)



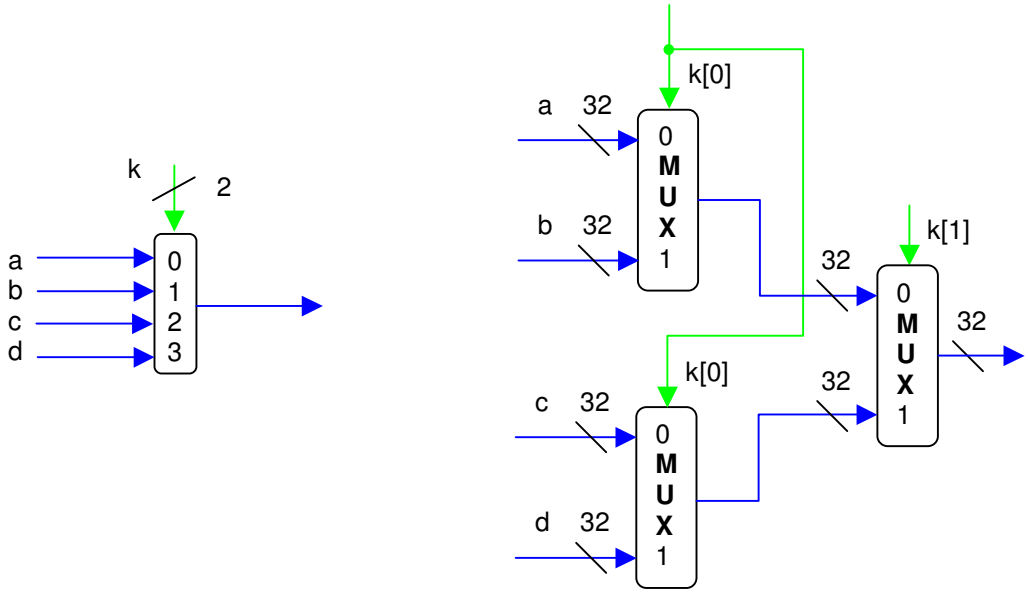
c)

d)



In a) ist links das Schaltungssymbol für ein 3-Weg AND Gatter gezeigt, rechts davon ist die Schaltung aus normalen AND-Gattern gezeigt. In b) finden Sie noch ein Beispiel für ein 8-Weg AND Gatter, links ist wieder das Schaltungssymbol gezeigt, rechts der Schaltungsaufbau aus normalen Gattern. Die Schaltungen für ein 3-Weg OR Gatter und ein 8-Weg OR Gatter sind in c) bzw. d) gezeigt.

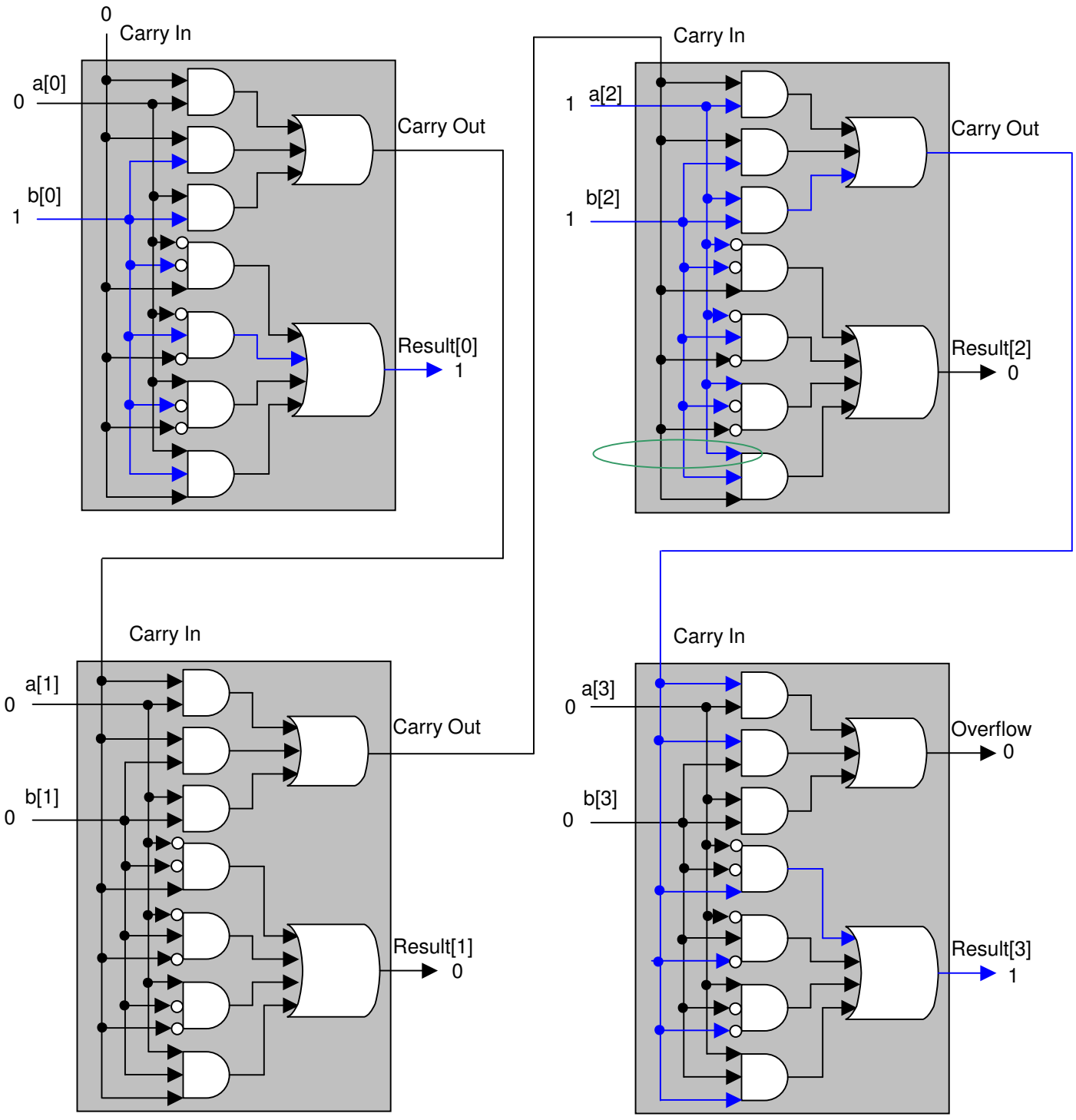
Aufgabe 1.7



Es ist links das Schaltungssymbol für den 4-Weg Multiplexer gezeigt, rechts davon der Aufbau aus den 2-Weg Multiplexern. Ist das Kontrollsignal k[0] gleich 0, so werden die Signale a und c durch die ersten beiden Multiplexer gelassen. Das Kontrollsignal k[1] entscheidet dann, ob a oder c. Für a ist das 2-Bit breite Kontrollsignal k also 00, für c ist es 10. Falls das Kontrollsignal k[0] gleich 1 ist, so werden die Signale b und d durch die ersten beiden Multiplexer geleitet. Dann wird wieder durch k[1] eines davon ausgewählt. Für Signal b ist k[1] gleich 0, für Signal d ist [1] gleich 1. Zusammengefasst:

Signal	k[1]	k[0]	Dezimalwert des Kontrollsignals
a	0	0	0
b	0	1	1
c	1	0	2
d	1	1	3

Aufgabe 1.8



Wir addieren a = 4: 0100 a[3] = 0, a[2] = 1 , a[1] = 0, a[0] = 0 und b = 5: 0101 b[3] = 0, b[2] = 1, b[1] = 0, b[0] = 1. Das erste „Carry In“ beim Least Significant Bit ist gleich 0. Die Signalleitungen mit einer 1 sind in der Schaltung blau hervorgehoben. An den Resultatausgängen liegen die Signale Result[3] = 1, Result[2] = 0, Result[1] = 0, Result[0] = 1 an. Also der Binäre Wert 1001: 9. Am Ausgangsignal Overflow = 0 sehen wir, dass das Resultat auch stimmt.

Aufgabe 1.9

- -5-2 ist gleichbedeutend mit (-5) + (-2). Die Zahlen im binären Zweierkomplement sind (-5): 1011, (-2): 1110. Nun die Addition: 1011 + 1110 = 11001. Das Resultat hat nur 4 Bit breite, das fünfte Signal wäre der Ausgang Overflow: 1(1001). Die 1001 entspricht im Zweierkomplement –7.

- 7-8 ist gleichbedeutend mit 7 + (-8). Die Zahlen im binären Zweierkomplement sind 7: 0111, (-8): 1000. Nun die Addition: 0111 + 1000 = 01111. Das Resultat hat 4 Bit breite, das fünfte Signal wäre der Ausgang Overflow: 0(1111). Die 1111 entspricht im Zweierkomplement –1.

- -8+4 ist gleichbedeutend mit (-8) + 4. Die Zahlen im binären Zweierkomplement sind (-8): 1000, 4: 0100. Nun die Addition: 1000 + 0100 = 01100. Das Resultat hat 4 Bit breite, das fünfte Signal wäre der Ausgang Overflow: 0(1100). Die 1100 entspricht im Zweierkomplement –4.

- -5+7 ist gleichbedeutend mit (-5) + 7. Die Zahlen im binären Zweierkomplement sind (-5): 1011, 7: 0111. Nun die Addition: 1011 + 0111 = 10010. Das Resultat hat 4 Bit breite, das fünfte Signal wäre der Ausgang Overflow: 1(0010). Die 0010 entspricht im Zweierkomplement +2.

Aufgabe 1.10

In a) ist die vollständige 4-Bit ALU gezeigt mit den Eingangssignalen a gleich 2: 0010 und b gleich 5: 0101. Das Kontrollsignal Subtraction ist 0, wir können also vor dem Multiplexer die Resultate für die Operationen AND, OR und Addition auslesen. Die Datenleitungen mit einer 1 als Signal sind blau hervorgehoben:

- Das Resultat für AND ist: Result[3] = 0, Result [2] = 0, Result[1] = 0, Result [0] = 0, also ist das 4-Bit breite Resultat: 0000.

- Das Resultat für OR ist: Result[3] = 0, Result[2] = 1, Result[1] = 1, Result [0] = 1, also ist das 4-Bit breite Resultat: 0111.

- Das Resultat für Addition ist: Result[3] = 0, Result[2] = 1, Result[1] = 1, Result [0] = 1, also ist das 4-Bit breite Resultat: 0111. Als Dezimalwert entspricht dies dem Wert 7.

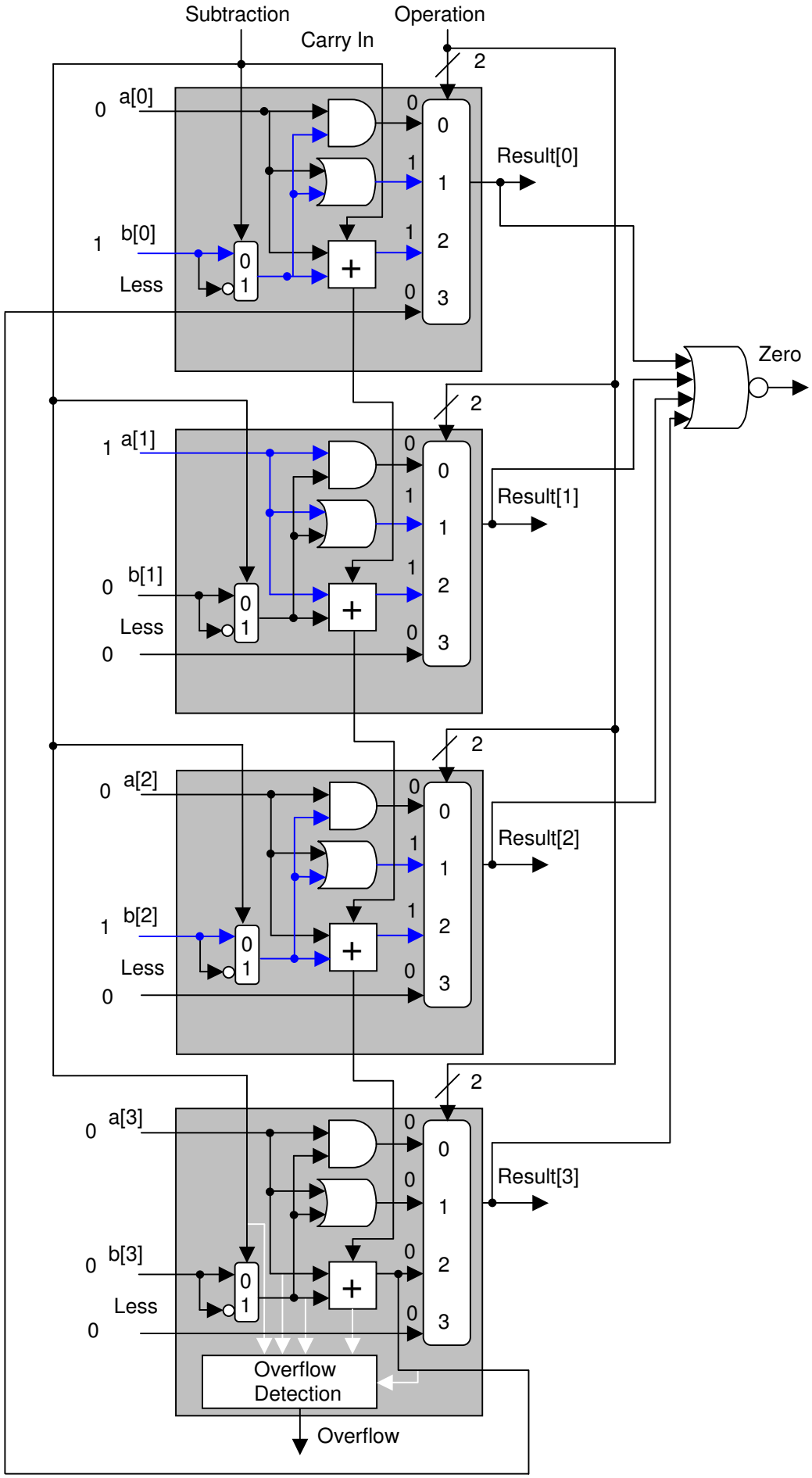
Für die Operationen Subtraktion und „set on less than“ wird das Kontrollsignal Subtraction = 1 benötigt. Dies ist in b) gezeichnet. Die Resultate für die Operationen können wieder vor dem Multiplexer abgelesen werden.

- Das Resultat für Subtraktion ist: Result[3] = 1, Result[2] = 1, Result[1] = 0, Result [0] = 1, also ist das 4-Bit breite Resultat: 1101. Als Dezimalwert entspricht dies dem Wert -3.

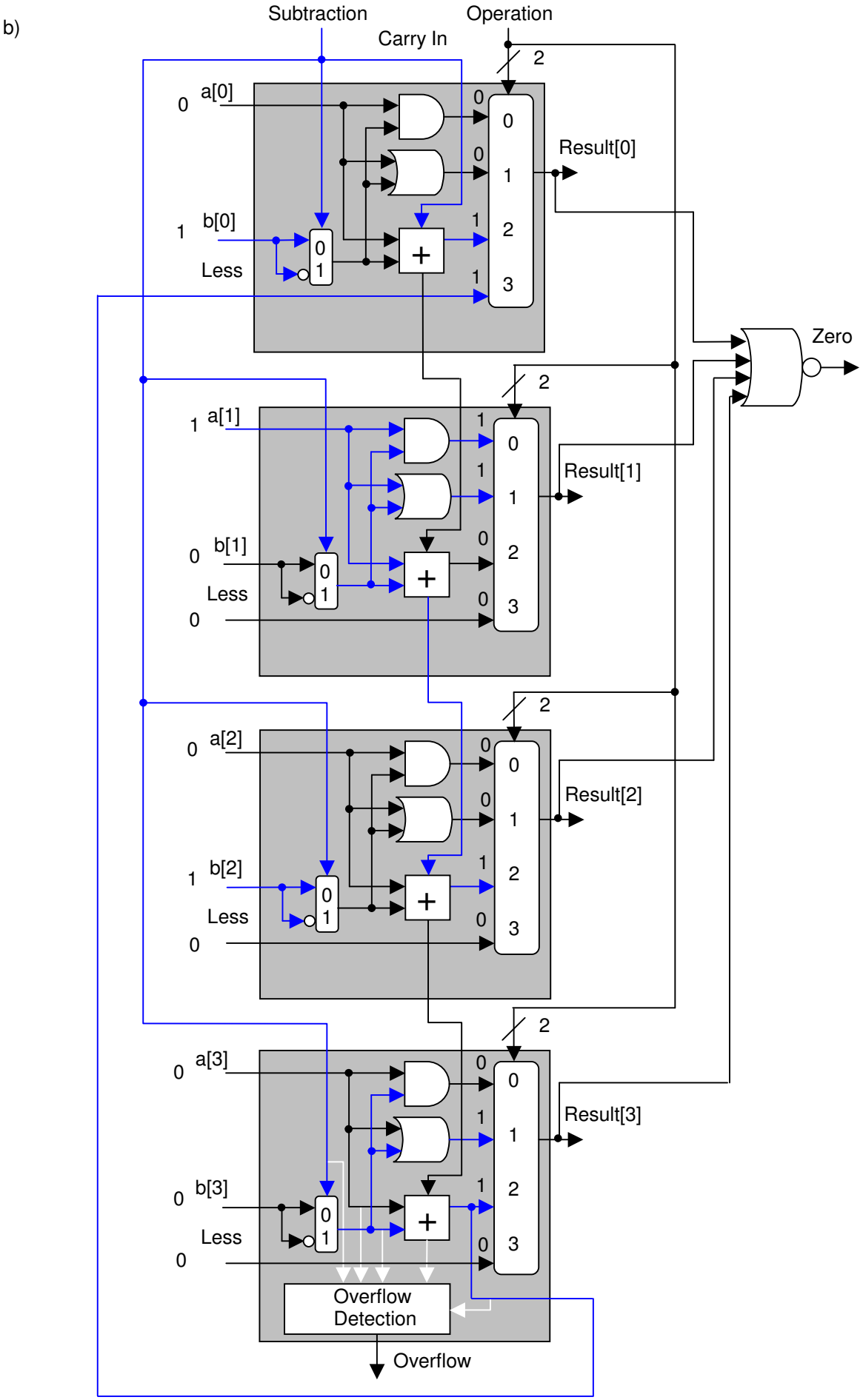
- Das Resultat für „set on less than“ ist: Result[3] = 0, Result[2] = 0, Result[1] = 0, Result [0] = 1, also ist das 4-Bit breite Resultat: 0001. Der Wert 2 ist also kleiner als der Wert 5.

Das Kontrollsignal „zero“ wurde soweit noch nicht betrachtet, aber es sollte leicht einzusehen sein, dass bei keiner Operation mit den Werten 2 und 5 das Kontrollsignal „zero“ auf 1 auswertet. Dies ist ja nur der Fall, falls die Werte identisch wären.

a)


Subtraction
Carry In
Operation
2
a[0]
0
b[0]
1
Less
Result[0]
Zero
a[1]
1
b[1]
0
Less
0
Result[1]
a[2]
0
b[2]
1
Less
0
Result[2]
a[3]
0
b[3]
0
Less
0
Result[3]
Overflow Detection
Overflow

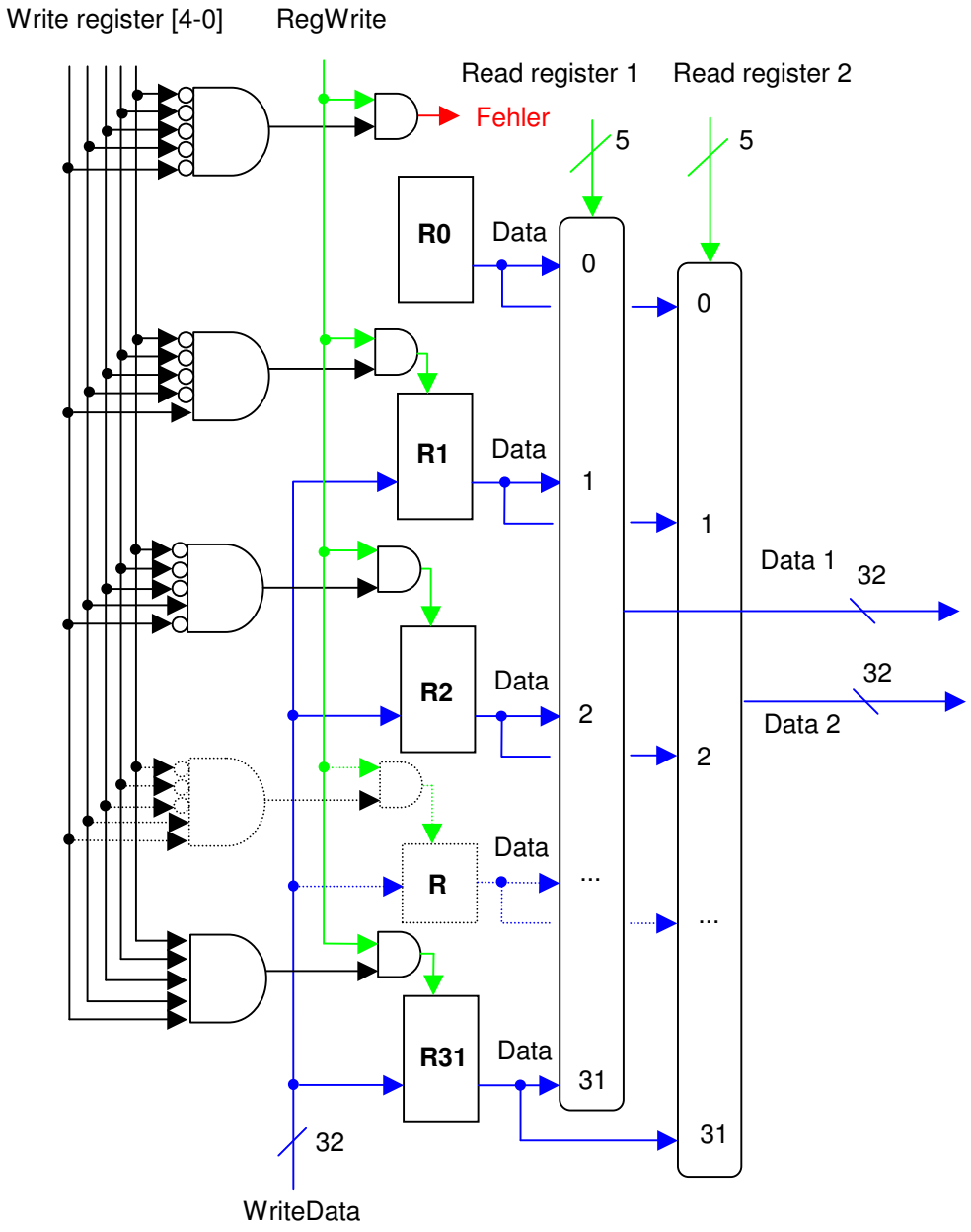
b)


Subtraction
Carry In
Operation
2
a[0]
b[0]
Less
Result[0]
a[1]
b[1]
Less
Result[1]
a[2]
b[2]
Less
Result[2]
a[3]
b[3]
Less
Result[3]
Zero
Overflow Detection
Overflow

Aufgabe 1.11

Aus den 32 einzelnen Register soll ein bestimmtes Signal an den Ausgang „Data 1“ geleitet werden. Dies kann mit einem 32-Weg Multiplexer erreicht werden. Dieser benötigt 5 Kontrollsignale, was genau der „Read register 1“ Adresse entspricht. Für den „Data 2“ Ausgang werden alle Ausgangssignale in einem zweiten parallelen 32-Weg Multiplexer ausgewählt, der mit den 5 Kontrollsignalen „Read register 2“ gesteuert wird.

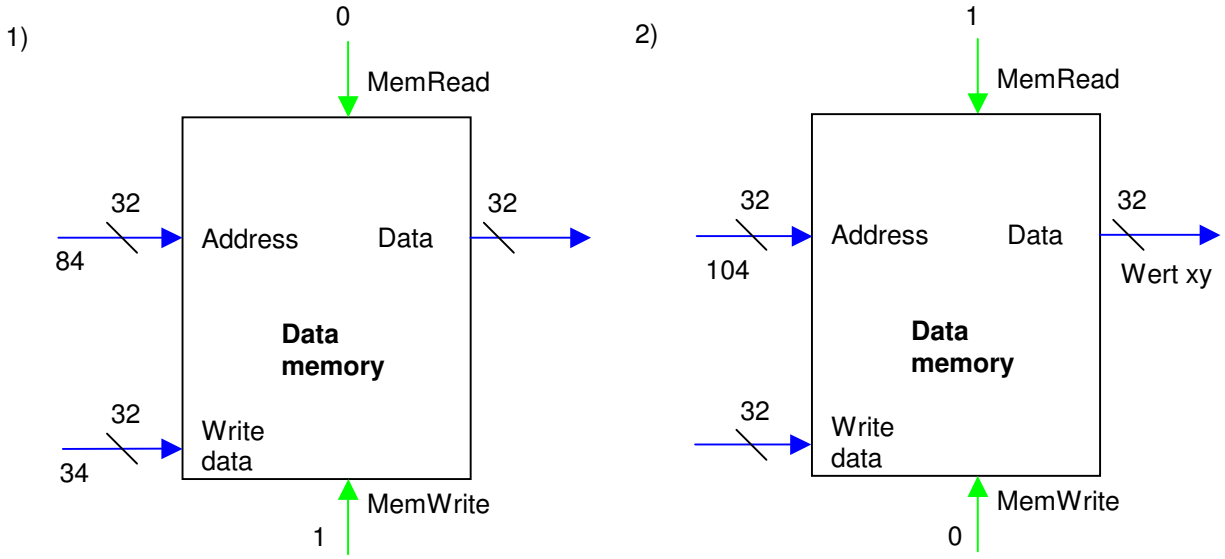
Die Leitungen des Eingangs „Write Data“ müssen alle Register erreichen können. Die Adresse des Eingangs „Write Register „ wählt nun ein bestimmtes Register aus. Mit 5-Weg AND Gattern und entsprechenden NOT Gattern können wir für jedes Register eine „Adresse“ zuweisen. Ist das Kontrollsignal „RegWrite“ gleich 1, so wird in das ausgewählte Register geschrieben. Achtung: Das erste Register R[0] ist immer 0, in dieses Register darf also nicht geschrieben werden. Das „RegWrite“ Signal zu diesem Register könnte einfach weggelassen werden, oder man kann das Signal für eine Fehlermeldung nutzen.



Aufgabe 1.7

a) In einem Clock-Zyklus wird immer ein Wert aus dem Instruction Memory gelesen. Die Instruktion könnte eine Anweisung sein, im Zyklus einen Wert aus dem Data Memory zu lesen. Diese Instruktion wird im selben Clock-Zyklus abgearbeitet. Ein Memory kann aber nicht in einem Zyklus zwei Werte ausgegeben. Um dies trotzdem zu erreichen, müssen zwei verschiedene Memory Bausteine verwendet werden.

b)



c) Ein Clock-Zyklus ist die maximale Dauer, die benötigt wird, bis die Schaltungssignale bei einer Instruktion alle Schaltungen und Komponenten durchwandert haben.

Ist der Clock-Zyklus nicht exakt regelmäßig, muss garantiert sein, dass bei einem minimalen Clock-Zyklus die Schaltungssignale alle Komponenten durchwandert haben. Der durchschnittliche Clock-Zyklus dauert deshalb länger, als bei einem idealen Clock-Zyklus.

Kommen bei einem Clock-Zyklus das Clock-Signal nicht an alle Komponenten gleichzeitig an, so ist einzuberechnen, bei welcher Komponente das Signal als letztes ankommt. Die Dauer, die das Clock-Signal später bei der letzten Komponente ankommt als bei der ersten Komponente, muss bei der realen Clock hinzuaddiert werden.

Aufgabe 2.1

a) Bei der Operation, die durch den Assembler-Befehl der Klasse R-Format ausgeführt werden soll, sind insgesamt 3 Register als Operanden beteiligt.
b) sub $t0, $t1, $t2

Aufgabe 2.2

a) Bei der Operation, die durch den Assembler-Befehl der Klasse I-Format ausgeführt werden soll, sind insgesamt 2 Register und eine Konstante als Operanden beteiligt.
b) lw $t0, 64($t1)

Aufgabe 2.3

a) Bei der Operation, die durch den Assembler-Befehl der Klasse J-Format ausgeführt werden soll, ist nur eine Konstante als Operand beteiligt.
b) jal TARGET

Aufgabe 2.4

- Branch on lower (bl):

```
bl $t0, t$1, LOOP       // falls der Wert im Register $t0 kleiner als der Wert im Register $t1
                        // ist, wird auf die Adresse des Labels „LOOP“ gesprungen
=

slt $t2, $t0, $t1               // falls $t0 kleiner als $t1 ist wird in $t2 eine 1 abgelegt
bne $zero, $t2, LOOP            // falls $t2 nicht 0 ist wird auf die Adresse des Labels „LOOP“
                                // verzweigt
```

- Branch on greater/equal (bge):

```
bge $t0, $t1, LOOP      // falls der Wert im Register $t0 grösser/gleich als der Wert im
                        // Register $t1 ist, dann wird auf die Adresse des Labels „LOOP“
                        // gesprungen
=

slt $t2, $t0, $t1               // falls $t0 kleiner als $t1 ist wird in $t2 eine 1 abgelegt
beq $zero, $t2, LOOP            // falls $t2  0 ist wird auf die Adresse des Labels „LOOP“
                                // verzweigt
```

- Increase (inc):

```
inc $t0,                // der Wert in Register $t0 wird mit 1 addiert und wieder im Register
                        // $t0 abgespeichert.
=

addi $t0, $t0, 1
```

- Decrease (dec):

```
dec $t0,                // der Wert in Register $t0 wird mit 1 subtrahiert und wieder im
                        // Register $t0 abgespeichert.
=

subi $t0, $t0, 1
```

Aufgabe 2.5

```
Mem[B+0] = Mem[A+0] + 100        // Addition eines Wertes aus dem Daten Memory mit einer
                                 // Konstanten in Pseudocode
```

=

```
lw $t0, 0($a0)                   // Lade den Wert bei Adresse A aus dem Memory in das Register $t0
addi $t0, $t0, 100               // Addiere den Werte aus Register $t0 und eine Konstante. Das
                                 // Resultat wird wieder in Register $t0 abgespeichert.
sw $t0, 0($a1)                   // Speichert den Wert aus Register $t0 im Data Memory unter der
                                 // Adresse B
```

Aufgabe2.6

```
Mem[A+5] = Mem[A+5] + 24         // Addition eines Arrayfeldes aus dem Daten Memory mit einer
                                 // Konstanten in Pseudocode
```

=

```
lw $t0, 20($a0)                  // Lade den Wert des fünften Feldes des Arrays A aus dem Memory in
                                 // das Register $t0
addi $t0, $t0, 24                // Addiere den Werte aus Register $t0 und eine Konstante. Das
                                 // Resultat wird wieder in Register $t0 abgespeichert.
sw $t0, 20($a0)                  // Speichert den Wert aus Register $t0 im fünften Feld des Arrays A.
```

Aufgabe 2.7

```
        switch k: {
                case k = 0: { d = x + y};
                case k = 1: { d = x – z};
                case k = 2: { d = y + z};
                case default: { d = y – z};
        };
```

=

```
         beq $a0 , $zero, CASE0;        // Falls k = 0, Sprung aufs Label CASE0
         addi $t0, $zero, 1;            // Register $t0 = 1
         beq $a0 , $t0, CASE1;          // Falls k = $t0 Sprung aufs Label CASE1
         addi $t0, $t0, 1 ;             // $t0 = $t0 + 1
         beq $a0 , $t0, CASE2;          // Falls k = $t0 Sprung aufs Label CASE2
         sub $v0, $a2, $a3;             // Case Default: d = y - z
         j END;                         // Sprung aufs Label END
CASE0: add $v0, $a1, $a2;               // Case 0: d = x + y
         j END;                         // Sprung aufs Label END
CASE1: sub $v0, $a1, $a3;               // Case 1: d = x - z
         j END;                         // Sprung aufs Label END
CASE2: add $v0, $a2, $a3;               // Case 2: d = y + z
         j END;                         // Sprung aufs Label END
END:     ….                             // Switch-Statement beendet
```

Aufgabe 2.8

```
i = 0;
do {
        A[i] = 0;
        i = i + 1;
}
while ( i < A.size)
```

=

```
        or $t0, $zero, $zero;          // i  = 0 in Register $t0
LOOP:   sw $zero, 0($a0);              // A[x] = 0
        addi $a0, $a0, 4;              // Adresse A[x+1] = A[x] + 4
        addi $t0, $t0, 1;              // i = i + 1
        bne $a1, $t0, LOOP             // Falls i = A.size wird der LOOP beendet
```

Aufgabe 2.9

a) Vor dem Aufbau eines neuen Stack-Frames für eine neue Prozedur gilt immer Stack-Pointer = Frame-Pointer. Dasselbe gilt auch, wenn eine Prozedur beendet wird und ihr Stack-Frame abgebaut wurde.

b) Es muss mindestens der Frame-Pointer und die Rücksprung-Adresse einer Prozedur auf dem Stack gespeichert werden. Bei unserer Calling Convention wird zwingend noch der Platz für die Register $s0-$s7 angelegt. Je nach Prozedur kommen noch die Zwischenresultate dazu, die nach dem neuen Prozedur-Aufruf wieder gebraucht werden. Dies können unter Umständen ganze Datenstrukturen sein. Wird an die neue Prozedur ein Parameter übergeben, der nicht in die Register $a0 - $a3 passt, oder werden mehr als 4 Parameter übergeben, so wird dies auch über den Stack gemacht. Dasselbe gilt, falls von der Prozedur ein Resultat erhalten werden soll, das nicht in die Register $v0 und $v1 passt.

c) Der Platz für die Register $s0-$s7 wird Aufgrund der Calling Convention angelegt, obwohl dies eigentlich nicht unbedingt notwendig wäre. Eine Begründung, warum dies so gemacht wird ist in Kapitel 6 angegeben. Man könnte aber problemlos eine Calling Convention definieren, in der dieses „Platz reservieren“ nicht gefordert wird.

Aufgabe 2.10

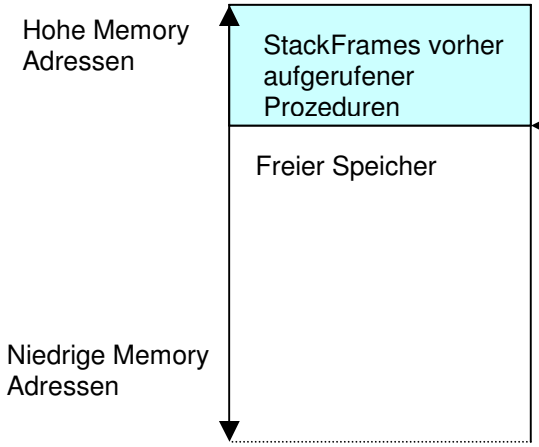
Die Prozedur nimmt einen Parameter a im Register $a0 entgegen. Als Resultat wird ein Wert im Register $v0 zurückgegeben. Bei dem Aufruf der Multiplikationsprozedur wird der Wert a zwischengespeichert. Die Stackbelegung ist auf der nächsten Seite abgebildet.

```
int Fakultät (int a){
        b = 1;
        while (a > 1){
                b = Multiplikation (b, a);
                a = a –1 ;
        }
        return b ;
}
```

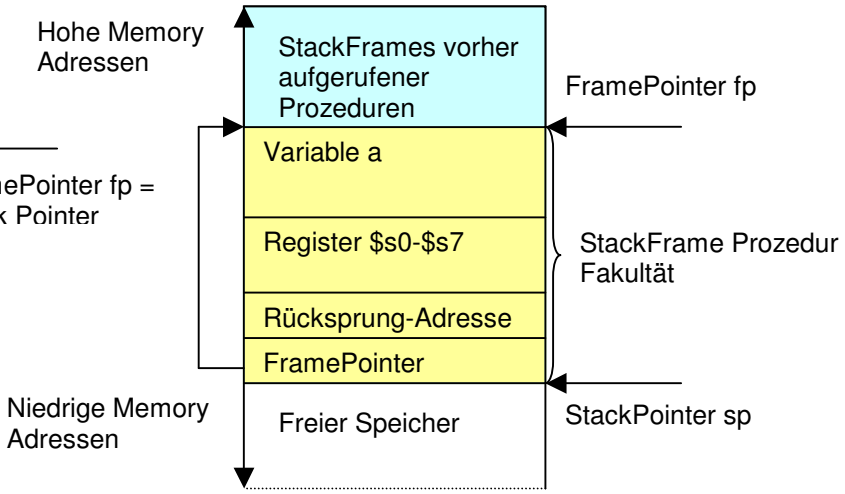
=

```
FAK :   ori $t0, $zero, 1 ;          // Variable b in Register $t0 mit 1 initialisiert
        or $t1, $a0, $zero ;         // Variable a aus $a0 in Register $t1 verschoben
LOOP : slti $t2, $t1, 2 ;            // Falls $t1 grösser/gleich 2 ist, dann ist  Register $t2 gleich 0
        bne $t2, $zero, END ;        // Falls $t2 nicht 0 ist, wird auf das Label END gesprungen
        subi $sp, $sp, 4;            // Vergrössert Stackframe für Variable a
        sw $t1, 0($sp);              // Speichert Variable a auf Stack
        subi $sp, $sp, 40;           // Vergrössert Stackframe für Save-Register, Rücksprung-
                                     // Adresse und Frame-Pointer
        sw $ra, -40($fp);            // Speichert Rücksprung-Adresse auf dem Stack
        sw $fp, -44($fp);            // Speichert Frame-Pointer auf dem Stack
        or $a0, $t0, $zero ;         // Argument-Variable b in Register $a0
        or $a1, $t1, $zero ;         // Argument-Variabe a in Register $a1
        or $fp, $sp, $zero;          // FramePointer = StackPointer
        jal MUL ;                    // Sprung auf die Prozedur MUL ;
        lw $fp, 4($sp);              // FramePointer wiederherstellen
        lw $ra, 8($sp);              // Rücksprung-Adresse wiederherstellen
        addi $sp, $sp, 40;           // StackFrame verkleinern
        lw $t1 0($fp);               // Variable a aus Stack in Register $t1
        or $sp, $fp, $zero;          // StackPointer = FramePointer. StackFrame wieder abgebaut
        or $t0, $v0, $zero;          // Ergebnis aus Prozedur MUL aus $v0 in Register $t0
        subi $t1, $t1, 1 ;           // Subtraktion der Variable a um 1
        j LOOP ;                     // Sprung auf das Label LOOP
END :   or $v0, $t0, $zero;          // Ergebnis der Prozedur aus $t0 in das Register $v0
        jr $ra;                      // Rücksprung aus der Prozedur FAK.
```

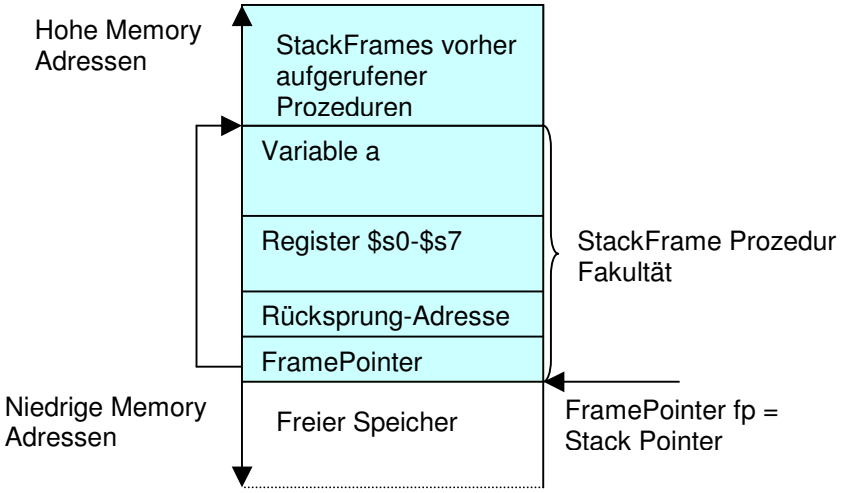
1)



2)



3)



Am Anfang ist wie bei 1) noch kein StackFrame für die Prozedur Fakultät angelegt. Die Prozedur Fakultät erstellt ein StackFrame, um die Variable a sowie um den Frame-Pointer und die Rücksprung-Adresse zu speichern wie in 2). Wenn die Prozedur Fakultät die Prozedur Multiplikation aufruft, wird wie in 3) für die Prozedur Multiplikation die Möglichkeit gegeben ein StackFrame anzulegen. Ist die Prozedur Multiplikation abgeschlossen wird zur Prozedur Fakultät zurückgesprungen. Von der Prozedur Fakultät wird das eigene StackFrame wieder ganz abgebaut. 1). Danach kann die Prozedur Fakultät wieder das StackFrame füllen, für den nächsten Aufruf der Prozedur Multiplikation. 2). Der Vorgang wiederholt sich nun für jeden while-Schleifendurchgang einmal.

Aufgabe 2.11

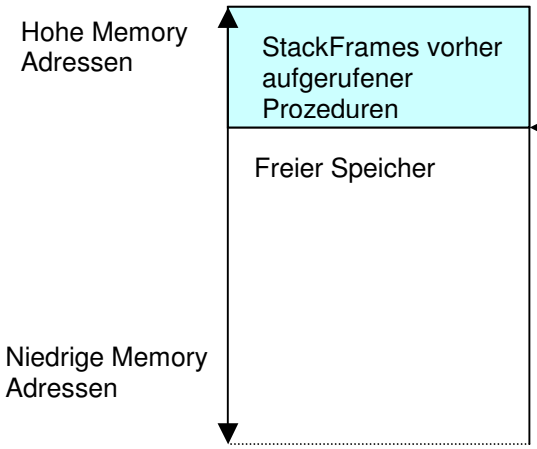
Die Prozedur nimmt einen Parameter x im Register $a0 entgegen. Als Resultat wird ein Wert im Register $v0 zurückgegeben. Bei dem Aufruf der Multiplikationsprozedur wird der Wert x zwischengespeichert. Die Stackbelegung ist auf der nächsten Seite abgebildet.

```
int Fakultät_R (int x){
        if (x == 1) return 1;
        else {
                y = Fakultät (x-1);
                return Multiplikation (x, y);
        };
} ;
```

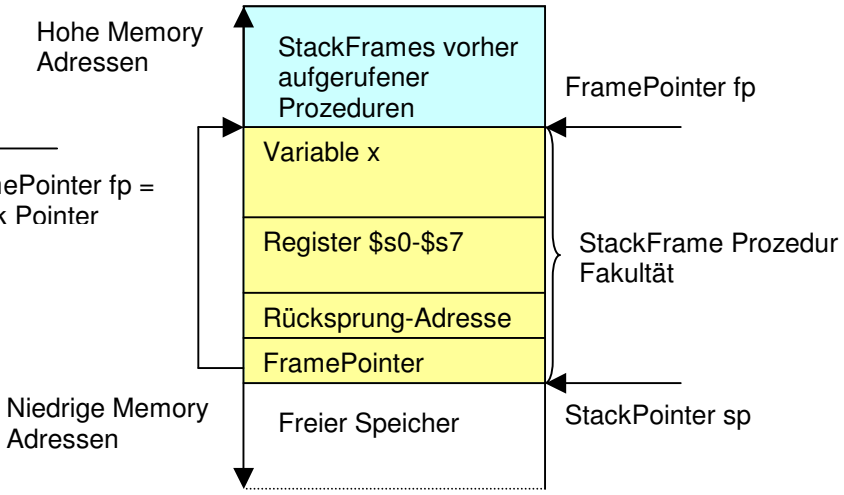
=

```
FAK_R :ori  $t0, $zero, 1          // Register $t0 wird mit 1 gefüllt
       bne $t0, $a0, ELSE          // Falls x nicht gleich 1, Sprung aufs Label ELSE
       or $v0, $t0, $zero          // x ist gleich 1. Resultat-Wert 1 in Register $v0 gespeichert
       j END;                      // Sprung aufs Label END
ELSE:  subi $sp, $sp, 4;           // Vergrössert Stackframe für Variable x
       sw $a0, 0($sp);             // Speichert Variable x auf Stack
       subi $sp, $sp, 40;          // Vergrössert Stackframe für Save-Register, Rücksprung-
                                   // Adresse und Frame-Pointer
       sw $ra, -40($fp);           // Speichert Rücksprung-Adresse auf dem Stack
       sw $fp, -44($fp);           // Speichert Frame-Pointer auf dem Stack
       subi $a0, $a0, 1;           // Argument-Wert x-1 in das Register $a0
       or $fp, $sp, $zero;         // FramePointer = StackPointer
       jal FAK_R ;                 // Sprung auf die Prozedur FAK_R
       lw $fp, 4($sp);             // FramePointer wiederherstellen
       lw $ra, 8($sp);             // Rücksprung-Adresse wiederherstellen
       addi $sp, $sp, 40;          // StackFrame verkleinern
       lw $t1 0(fp);               // Variable x aus Stack in Register $t1
       or $sp, $fp, $zero;         // StackPointer = FramePointer. StackFrame wieder abgebaut
       or $t2, $v0, $zero;         // Ergebnis aus Prozedur FAK_R aus $v0 in Register $t2
       subi $sp, $sp, 40;          // Vergrössert Stackframe für Save-Register, Rücksprung-
                                   // Adresse und Frame-Pointer
       sw $ra, -40($fp);           // Speichert Rücksprung-Adresse auf dem Stack
       sw $fp, -44($fp);           // Speichert Frame-Pointer auf dem Stack
       or $a0, $t1, $zero;         // Argument-Wert x in das Register $a0
       or $a1, $t2, $zero;         // Argument-Wert y in das Register $a1
       or $fp, $sp, $zero;         // FramePointer = StackPointer
       jal MUL ;                   // Sprung auf die Prozedur MUL
       lw $fp, 4($sp);             // FramePointer wiederherstellen
       lw $ra, 8($sp);             // Rücksprung-Adresse wiederherstellen
       addi $sp, $sp, 40;          // StackFrame verkleinern
       or $sp, $fp, $zero;         // StackPointer = FramePointer. StackFrame wieder abgebaut
                                   // Ergebnis aus Prozedur MUL bleibt in $v0, da dies ja gerade
                                   // das Ergebnis der Prozedur FAK_R ist.
END:   jr $ra                      // Prozedur FAK_R beendet.
```

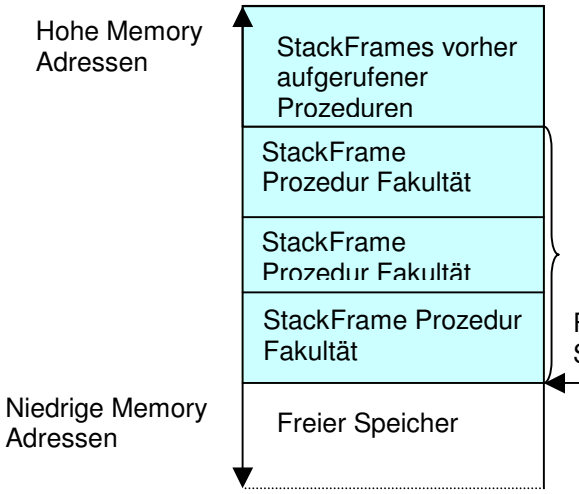
1)



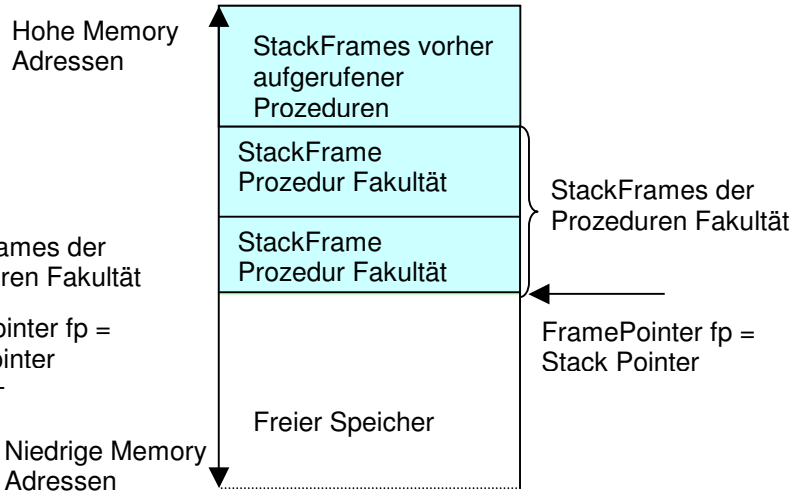
2)



3)



4)



Ist das Prozedur-Argument a nicht gleich 1, so wird am Anfang wird bei der Prozedur Fakultät ein StackFrame angelegt und der Wert der Variable x sowie die Rücksprung-Adresse und FramePointer auf dem Stack zwischengespeichert. 2) Danach wird wieder die Prozedur Fakultät mit dem Wert x-1 aufgerufen und ein neues StackFrame angelegt. Dies wiederholt sich solange, bis das Argument x gleich 1 ist. 3). Ist das Argument x gleich 1, so wird als Ergebnis eine 1 an die zuletzt aufgerufene Fakultätsprozedur geliefert. Diese kann nun die Multiplikationsprozedur aufrufen und wieder ein Ergebnis an die vorherige Fakultätsprozedur liefern. Ein Fakultätsprozedur-StackFrame wird abgebaut. 4). Wieder kann die Fakultätsprozedur die Multiplikationsprozedur aufrufen und ein Ergebnis berechnen. Wieder kann ein StackFrame abgebaut werden. Dies wiederholt sich so lange, bis alle Fakultätsprozedur-StackFrames abgebaut sind und ein schlussendliches Ergebnis zurückgeliefert werden kann.

Aufgabe 3.1

Die aktuelle Instruktionsadresse ist PC+4 ist 2’416’000’000: 10010000000000010011110000000000.

Die Adresse A = 2’415’910’000 ist in Binär : 10001111111111111101110001110000.

Die Adresse B= 2’415’990’000 ist in Binär: 10010000000000010001010011110000.

Die Adresse C= 2’600’000’000 ist in Binär: 10011010111110001101101000000000.

Die Adresse D= 2’690’000’000 ist in Binär: 10100000010101100010010010000000.

Die 4 Most Significant Bits der Instruktionsadresse müssen gleich den 4 Most Significant Bits der Sprungadresse sein. Dies ist bei Adresse B und C der Fall:

Adresse B: [1001]00000000000100010100111110000.
Es werden die beiden least Significant Bits abgeschnitten: [1001]00000000000100010100111100[00].
Der Rest ist die benötigte 26-Bit Konstante: 00000000000100010100111100.

Adresse C: [1001]10101111100011011010000000000.
Es werden die beiden least Significant Bits abgeschnitten: [1001]10101111100011011010000000[00].
Der Rest ist die benötigte 26-Bit Konstante: 10101111100011011010000000.

Die Adressen A und D sind in einem so nicht erreichbaren Memory-Bereich.

Aufgabe 3.2

Die aktuelle Instruktionsadresse ist PC+4 ist 2’416’000’000: 10010000000000010011110000000000.
Das Zweierkomplement von PC + 4 ist: 01101111111111101100010000000000.

- Die Adresse A = 2’415’910’000 ist in Binär : 10001111111111111101110001110000.

A - (PC + 4):

```
                10001111111111111101110001110000
        +       01101111111111101100010000000000
        =       11111111111111101010000001110000

Shift right 2:  11111111111111111010100000011100
- 16 MSB:       1010100000011100
```

- Die Adresse B= 2’415’990’000 ist in Binär: 10010000000000010001010011110000.

B - (PC + 4):

```
                10010000000000010001010011110000
        +       01101111111111101100010000000000
        =       11111111111111111101100011110000

Shift right 2:  11111111111111111111011000111100
- 16 MSB:       1111011000111100
```

- Die Adresse C= 2’600’000’000 ist in Binär: 10011010111110001101101000000000.

C - (PC + 4):

```
                10011010111110001101101000000000
        +       01101111111111101100010000000000
        =       00001010111101111001111000000000

Shift right 2:  00000010101111011110011110000000
```

Durch das Kürzen der 16 Most Significant Bits wird mehr als nur das Vorzeichen abgeschnitten. Die Sprung-Adresse ist also in einem so nicht erreichbaren Memory-Bereich

- Die Adresse D= 2’690’000’000 ist in Binär: 10100000010101100010010010000000.

D - (PC + 4):

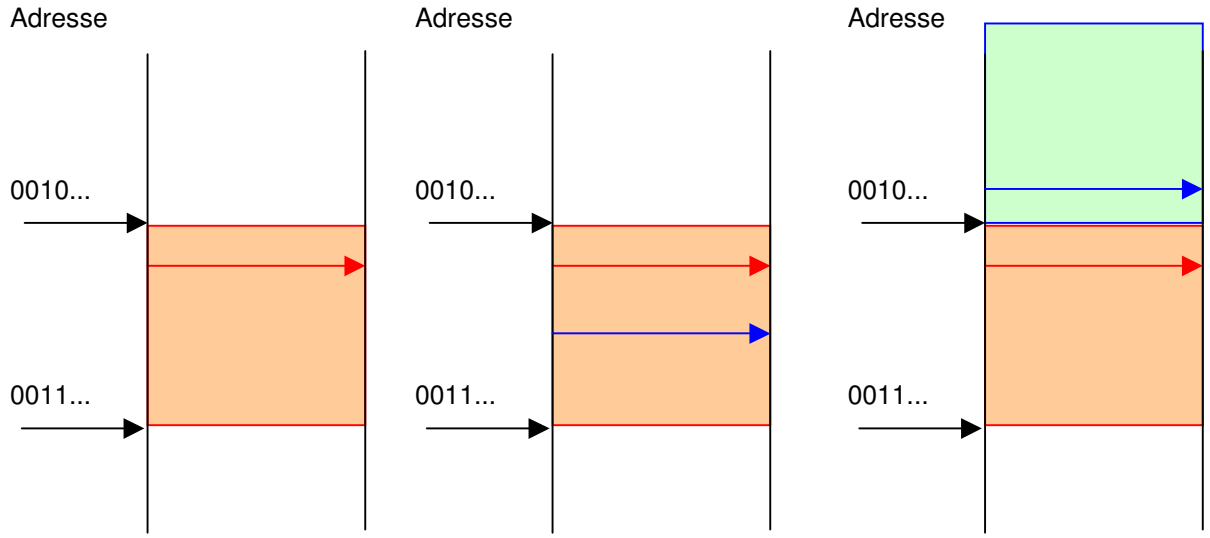
```
                10100000010101100010010010000000
        +       01101111111111101100010000000000
        =       00010000010101001110100010000000

Shift right 2:  00000100000101010011101000100000
```

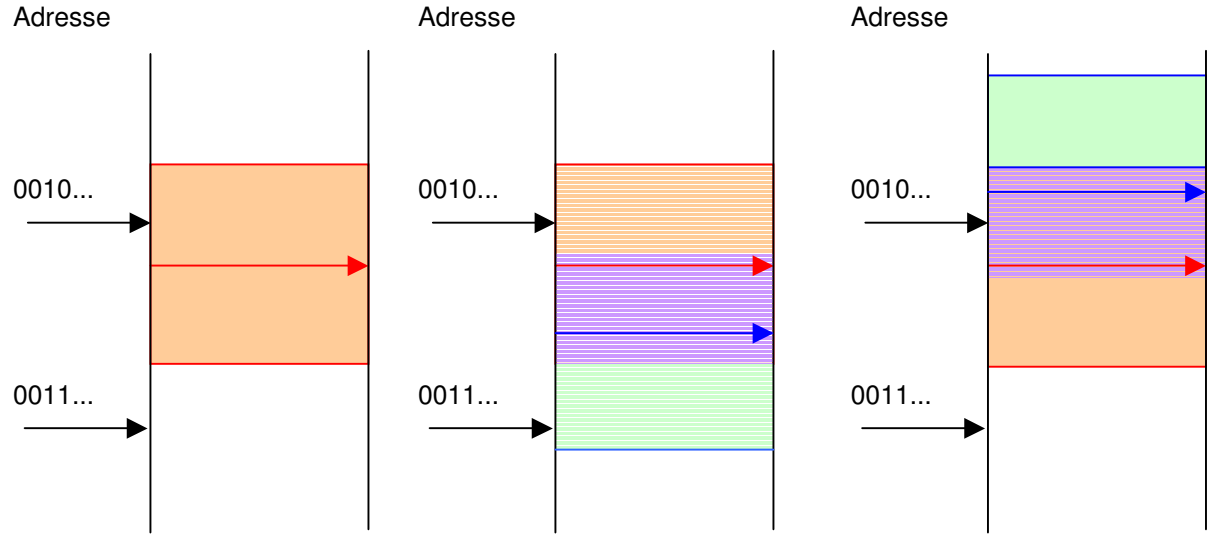
Durch das Kürzen der 16 Most Significant Bits wird mehr als nur das Vorzeichen abgeschnitten. Die Sprung-Adresse ist also wieder in einem so nicht erreichbaren Memory-Bereich.

Aufgabe 3.3

Jump und Jump and Link:



Branch on equal und Branch on not equal:



Bei einem J-Format Instruktionssprung, wird ein absoluter Bereich abhängig vom aktuellen PC adressiert. Eine kleine Änderung beim aktuellen PC ändert beim adressierbaren Bereich entweder gar nichts, oder es wird ein ganz neuer Bereich adressiert, wenn sich der Wert in den Most Significant Bit des PC geändert hat.

Bei den I-Format Sprüngen im Instruction Memory ist der adressierbare Bereich immer relativ zum aktuellen PC, und eine Verschiebung des PC führt zu einer Mitverschiebung des Bereiches.

Aufgabe 3.4

Wir haben den Opcode mit 6 Bit Breite, 2 hoch 6 sind 64 verschiedene Signale, also 64 verschiedene Instruktionen. Eine dieser Instruktionen haben wir zur Identifizierung der Klasse R-Format verwendet, also 63 Instruktionen und eine Klasse.

Die Klasse unterscheidet innerhalb der Klassenfelder „funct“ und „shamt“ mit insgesamt 11 Bits eine bestimmte Instruktion der Klasse, also 2 hoch 11 macht weitere 2048 Instruktionen.

Total: 63 + 2048 = 2111 Instruktionen.

Aufgabe 3.5

Die Prozedur „main“ könnte im Pseudocode wie folgt geschrieben werden:

```
void main (){
        f = 5;
        g = Fakultät(f);
        h = Fakultät_R(g);
}
```

=

```
MAIN:  ori $a0, $zero, 5;          // Argument-Register $a0 mit dem Wert 5 belegt
       or $sp, $fp, $zero;         // StackPointer = FramePointer. Stackpointer initialisiert
       subi $sp, $sp, 40;          // Vergrössert Stackframe für Save-Register, Rücksprung-
                                   // Adresse  und Frame-Pointer
       sw $ra, -40($fp);           // Speichert Rücksprung-Adresse auf dem Stack
       sw $fp, -44($fp);           // Speichert Frame-Pointer auf dem Stack
       or $fp, $sp, $zero;         // FramePointer = StackPointer
       jal FAK ;                   // Sprung auf die Prozedur FAK
       lw $fp, 4($sp);             // FramePointer wiederherstellen
       lw $ra, 8($sp);             // Rücksprung-Adresse wiederherstellen
       addi $sp, $sp, 40;          // StackFrame verkleinern
       or $sp, $fp, $zero;         // StackPointer = FramePointer. StackFrame wieder abgebaut
       or $a0, $v0, $zero;         // Ergebnis aus Prozedur FAK aus $v0 in Register $a0
       subi $sp, $sp, 40;          // Vergrössert Stackframe für Save-Register, Rücksprung-
                                   // Adresse  und Frame-Pointer
       sw $ra, -40($fp);           // Speichert Rücksprung-Adresse auf dem Stack
       sw $fp, -44($fp);           // Speichert Frame-Pointer auf dem Stack
       or $a0, $t1, $zero;         // Argument-Wert g in das Register $a0
       or $fp, $sp, $zero;         // FramePointer = StackPointer
       jal FAK_R ;                 // Sprung auf die Prozedur FAK_R
       lw $fp, 4($sp);             // FramePointer wiederherstellen
       lw $ra, 8($sp);             // Rücksprung-Adresse wiederherstellen
       addi $sp, $sp, 40;          // StackFrame verkleinern
       or $sp, $fp, $zero;         // StackPointer = FramePointer. StackFrame wieder abgebaut
                                   // Ergebnis aus Prozedur FAK_R  bleibt in $v0
       jr $ra                      // Prozedur MAIN beendet.
```

Die Instruktionen der Prozedur “main” kommen im Instruction Memory an die Adresse 0 zu liegen. Die Prozedur „Fakultät_R“ wird daran anschliessend gespeichert, gefolgt von der Prozedur „Fakultät“. Wir definieren noch die Rücksprung-Adresse für die Prozedur „main“ im Register $ra als 0. Das Programm „main“ wird also endlos wiederholt. Der Frame-Pointer wird im Register $fp mit der höchsten Data Memory Adresse initialisiert, da der Stack ja von „oben nach unten“ wächst.

Assembler Befehl		Adresse	Instruction Memory	
MAIN:	ori $a0, $zero, 5;	0000	MAIN:	001011 00100 00000 0000000000000101
	or $sp, $fp, $zero;	0004		100000 11101 11110 00000 00000 000010
	subi $sp, $sp, 40;	0008		001001 11101 11101 0000000000101000
	sw $ra, -40($fp);	0012		001110 11111 11110 1111111111011000
	sw $fp, -44($fp);	0016		001110 11110 11110 1111111111010100
	or $fp, $sp, $zero;	0020		100000 11110 11101 00000 00000 000010
	jal FAK ;	0024		000111 FAK
	lw $fp, 4($sp);	0028		001101 11110 11101 0000000000000100
	lw $ra, 8($sp);	0032		001101 11111 11101 0000000000001000
	addi $sp, $sp, 40;	0036		001000 11101 11101 0000000000101000
	or $sp, $fp, $zero;	0040		100000 11101 11110 00000 00000 000010
	or $a0, $v0, $zero;	0044		100000 00100 00010 00000 00000 000010
	subi $sp, $sp, 40;	0048		001001 11101 11101 0000000000101000
	sw $ra, -40($fp);	0052		001110 11111 11110 1111111111011000
	sw $fp, -44($fp);	0056		001110 11110 11110 1111111111010100
	or $a0, $t1, $zero;	0060		100000 00100 01001 00000 00000 000010
	or $fp, $sp, $zero;	0064		100000 11110 11101 00000 00000 000010
	jal FAK_R ;	0068		000111 FAK_R
	lw $fp, 4($sp);	0072		001101 11110 11101 0000000000000100
	lw $ra, 8($sp);	0076		001101 11111 11101 0000000000001000
	addi $sp, $sp, 40;	0080		001000 11101 11101 0000000000101000
	or $sp, $fp, $zero;	0084		100000 11101 11110 00000 00000 000010
	jr $ra	0088		100000 11111 00000 00000 00000 001000
FAK_R	:ori $t0, $zero, 1	0092	FAK_R:	001011 01000 00000 0000000000000001
	bne $t0, $a0, ELSE	0096		010001 01000 00100 ELSE
	or $v0, $t0, $zero	0100		100000 00010 01000 00000 00000 000010
	j END1;	0104		000110 END1
ELSE:	subi $sp, $sp, 4;	0108	ELSE:	001001 11101 11101 0000000000000100
	sw $a0, 0($sp);	0112		001110 00100 11101 0000000000000000
	subi $sp, $sp, 40;	0116		001001 11101 11101 0000000000101000
	sw $ra, -40($fp);	0120		001110 11111 11110 1111111111011000
	sw $fp, -44($fp);	0124		001110 11110 11110 1111111111010100
	subi $a0, $a0, 1;	0128		001001 00100 00100 0000000000000001
	or $fp, $sp, $zero;	0132		100000 11110 11101 00000 00000 000010
	jal FAK_R ;	0136		000111 FAK_R
	lw $fp, 4($sp);	0140		001101 11110 11101 0000000000000100
	lw $ra, 8($sp);	0144		001101 11111 11101 0000000000001000
	addi $sp, $sp, 40;	0148		001000 11101 11101 0000000000101000
	lw $t1 0($fp);	0152		001101 01001 11110 0000000000000000
	or $sp, $fp, $zero;	0156		100000 11101 11110 00000 00000 000010
	or $t2, $v0, $zero;	0160		100000 01010 00010 00000 00000 000010
	subi $sp, $sp, 40;	0164		001001 11101 11101 0000000000101000
	sw $ra, -40($fp);	0168		001110 11111 11110 1111111111011000
	sw $fp, -44($fp);	0172		001110 11110 11110 1111111111010100
	or $a0, $t1, $zero;	0176		100000 00100 01001 00000 00000 000010
	or $a1, $t2, $zero;	0180		100000 00101 01010 00000 00000 000010
	or $fp, $sp, $zero;	0184		100000 11110 11101 00000 00000 000010
	jal MUL ;	0188		000111 MUL
	lw $fp, 4($sp);	0192		001101 11110 11101 0000000000000100
	lw $ra, 8($sp);	0196		001101 11111 11101 0000000000001000
	addi $sp, $sp, 40;	0200		001000 11101 11101 0000000000101000
	or $sp, $fp, $zero;	0204		100000 11101 11110 00000 00000 000010
END1:	jr $ra	0208	END1:	100000 11111 00000 00000 00000 001000

```
FAK :   ori $t0, $zero, 1 ;          0212
        or $t1, $a0, $zero ;         0216
LOOP1: slti $t2, $t1, 2 ;            0220
        bne $t2, $zero, END2;        0224
        subi $sp, $sp, 4;            0228
        sw $t1, 0($sp);              0232
        subi $sp, $sp, 40;           0236
        sw $ra, -40($fp);            0240
        sw $fp, -44($fp);            0244
        or $a0, $t0, $zero ;         0248
        or $a1, $t1, $zero ;         0252
        or $fp, $sp, $zero;          0256
        jal MUL ;                    0260
        lw $fp, 4($sp);              0264
        lw $ra, 8($sp);              0268
        addi $sp, $sp, 40;           0272
        lw $t1 0($fp);               0276
        or $sp, $fp, $zero;          0280
        or $t0, $v0, $zero;          0284
        subi $t1, $t1, 1 ;           0288
        j LOOP1 ;                    0292
END2:   or $v0, $t0, $zero;          0296
        jr $ra;                      0300


MUL:    or $t0, $zero, $zero         0304
LOOP:   slti $t1, $a1, 1             0308
        bne $t1, $zero, END          0312
        add $t0, $t0, $a0            0316
        subi $a1, $a1, 1             0320
        j LOOP                       0324
END:    or $v0, $t0, $zero           0328
        jr $ra                       0332
```

```
FAK:    001011 01000 00000 0000000000000001
        100000 01001 00100 00000 00000 000010
LOOP1:001100 01010 01001 0000000000000010
        010001 01010 00000 END2
        001001 11101 11101 0000000000000100
        001110 01001 11101 0000000000000000
        001001 11101 11101 0000000000101000
        001110 11111 11110 1111111111011000
        001110 11110 11110 1111111111010100
        100000 00100 01000 00000 00000 000010
        100000 00101 01001 00000 00000 000010
        100000 11110 11101 00000 00000 000010
        000111 MUL
        001101 11110 11101 0000000000000100
        001101 11111 11101 0000000000001000
        001000 11101 11101 0000000000101000
        001101 01001 11110 0000000000000000
        100000 11101 11110 00000 00000 000010
        100000 01000 00010 00000 00000 000010
        001001 01001 01001 0000000000000001
        000110 LOOP1
END2:   100000 00010 01000 00000 00000 000010
        100000 11111 00000 00000 00000 001000


MUL:    100000 01000 00000 00000 00000 000010
LOOP:   001100 01001 00101 0000000000000001
        010001 01001 00000 END
        100000 01000 01000 00100 00000 000100
        001001 00101 00101 0000000000000001
        000110 LOOP
END:    100000 00010 01000 00000 00000 000010
        100000 11111 00000 00000 00000 001000
```

Wir haben im Maschinencode insgesamt 11 Label, deren Sprung-Adressen noch auszurechnen sind.

- Bei Adresse 0024 soll auf die Zeile mit Adresse 0212 gesprungen werden. Es wird ein unbedingter Sprung mit dem Befehl „Jump and Link“ dafür benutzt.

PC + 4 ist 0028:	00000000000000000000000000011100
Sprungziel-Adresse ist 0212:	00000000000000000000000011010100
-4 Most Significant Bits	[0000]00000000000000000000011010100
-2 Least Significant Bits	[0000]00000000000000000000110101[00]
ergiebt die Konstante	00000000000000000000110101

- Bei Adresse 0068 soll auf die Zeile mit Adresse 0092 gesprungen werden. Es wird ein unbedingter Sprung mit dem Befehl „Jump and Link“ dafür benutzt.

PC + 4 ist 0072:	00000000000000000000000001001000
Sprungziel-Adresse ist 0092:	00000000000000000000000001011100
-4 Most Significant Bits	[0000]00000000000000000000001011100
-2 Least Significant Bits	[0000]00000000000000000000010111[00]
ergiebt die Konstante	00000000000000000000010111

- Bei Adresse 0096 soll auf die Zeile mit Adresse 0108 gesprungen werden. Es wird ein bedingter Sprung mit dem Befehl „Branch on not equal“ dafür benutzt.

PC + 4 ist 0100:	00000000000000000000000001100100
Zweierkomplement von PC+4	11111111111111111111111110011100
Sprungziel-Adresse ist 0108:	00000000000000000000000001101100
Addition	00000000000000000000000000001000
Shift right 2	00000000000000000000000000000010
-16 Most Significant Bits	0000000000000010

- Bei Adresse 0104 soll auf die Zeile mit Adresse 0208 gesprungen werden. Es wird ein unbedingter Sprung mit dem Befehl „Jump“ dafür benutzt.

PC + 4 ist 0108:	00000000000000000000000001101100
Sprungziel-Adresse ist 0208:	00000000000000000000000011010000
-4 Most Significant Bits	[0000]00000000000000000000011010000
-2 Least Significant Bits	[0000]00000000000000000000110100[00]
ergiebt die Konstante	00000000000000000000110100

- Bei Adresse 0136 soll auf die Zeile mit Adresse 0092 gesprungen werden. Es wird ein unbedingter Sprung mit dem Befehl „Jump and Link“ dafür benutzt.

PC + 4 ist 0140:	00000000000000000000000010001100
Sprungziel-Adresse ist 0092:	00000000000000000000000001011100
-4 Most Significant Bits	[0000]00000000000000000000001011100
-2 Least Significant Bits	[0000]00000000000000000000010111[00]
ergiebt die Konstante	00000000000000000000010111

- Bei Adresse 0188 soll auf die Zeile mit Adresse 0304 gesprungen werden. Es wird ein unbedingter Sprung mit dem Befehl „Jump and Link“ dafür benutzt.

PC + 4 ist 0192:	00000000000000000000000011000000
Sprungziel-Adresse ist 0304:	00000000000000000000000100110000
-4 Most Significant Bits	[0000]00000000000000000000100110000
-2 Least Significant Bits	[0000]00000000000000000001001100[00]
ergiebt die Konstante	00000000000000000001001100

- Bei Adresse 0224 soll auf die Zeile mit Adresse 0296 gesprungen werden. Es wird ein bedingter Sprung mit dem Befehl „Branch on not equal“ dafür benutzt.

PC + 4 ist 0228:	00000000000000000000000011100100
Zweierkomplement von PC+4	11111111111111111111111100011100
Sprungziel-Adresse ist 0296:	00000000000000000000000100101000
Addition	00000000000000000000000000001000100
Shift right 2	00000000000000000000000000010001
-16 Most Significant Bits	0000000000010001

- Bei Adresse 0260 soll auf die Zeile mit Adresse 0304 gesprungen werden. Es wird ein unbedingter Sprung mit dem Befehl „Jump and Link“ dafür benutzt.

PC + 4 ist 0264:	00000000000000000000000100001000
Sprungziel-Adresse ist 0304:	00000000000000000000000100110000
-4 Most Significant Bits	[0000]00000000000000000000100110000
-2 Least Significant Bits	[0000]00000000000000000001001100[00]
ergiebt die Konstante	00000000000000000001001100

- Bei Adresse 0292 soll auf die Zeile mit Adresse 0220 gesprungen werden. Es wird ein unbedingter Sprung mit dem Befehl „Jump“ dafür benutzt.

PC + 4 ist 0296:	00000000000000000000000100101000
Sprungziel-Adresse ist 0220:	00000000000000000000000011011100
-4 Most Significant Bits	[0000]00000000000000000000011011100
-2 Least Significant Bits	[0000]00000000000000000000110111[00]
ergiebt die Konstante	00000000000000000000110111

- Bei Adresse 0312 soll auf die Zeile mit Adresse 0328 gesprungen werden. Es wird ein bedingter Sprung mit dem Befehl „Branch on not equal“ dafür benutzt.

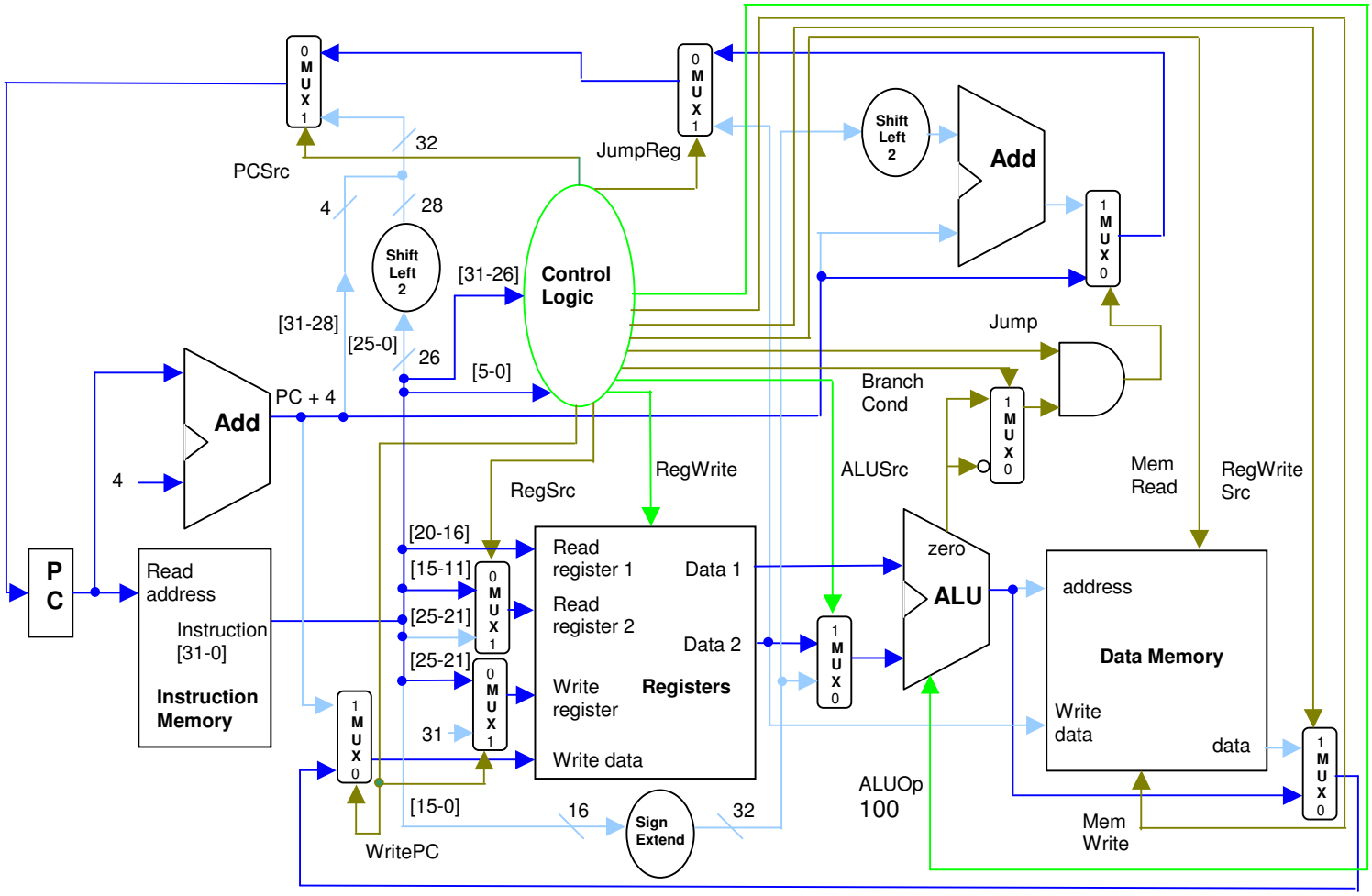
PC + 4 ist 0316:	00000000000000000000000100111100
Zweierkomplement von PC+4	11111111111111111111111011000100
Sprungziel-Adresse ist 0328:	00000000000000000000000101001000
Addition	00000000000000000000000000001100
Shift right 2	00000000000000000000000000000011
-16 Most Significant Bits	0000000000000011

- Bei Adresse 0324 soll auf die Zeile mit Adresse 0308 gesprungen werden. Es wird ein unbedingter Sprung mit dem Befehl „Jump“ dafür benutzt.

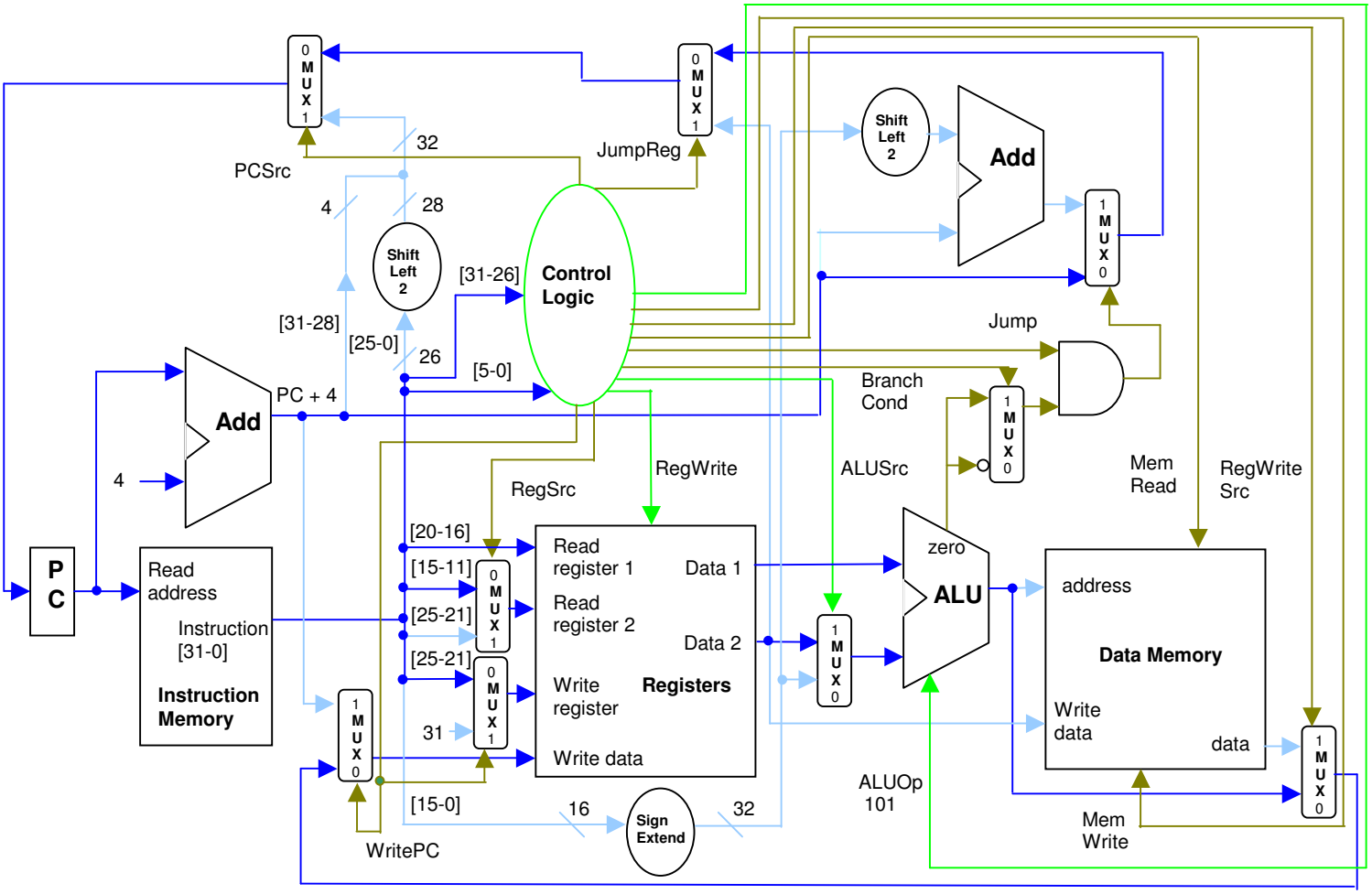
PC + 4 ist 0328:	00000000000000000000000101001000
Sprungziel-Adresse ist 0308:	00000000000000000000000100110100
-4 Most Significant Bits	[0000]00000000000000000000100110100
-2 Least Significant Bits	[0000]00000000000000000001001101[00]
ergiebt die Konstante	00000000000000000001001101

Aufgabe 4.4 / 5.5

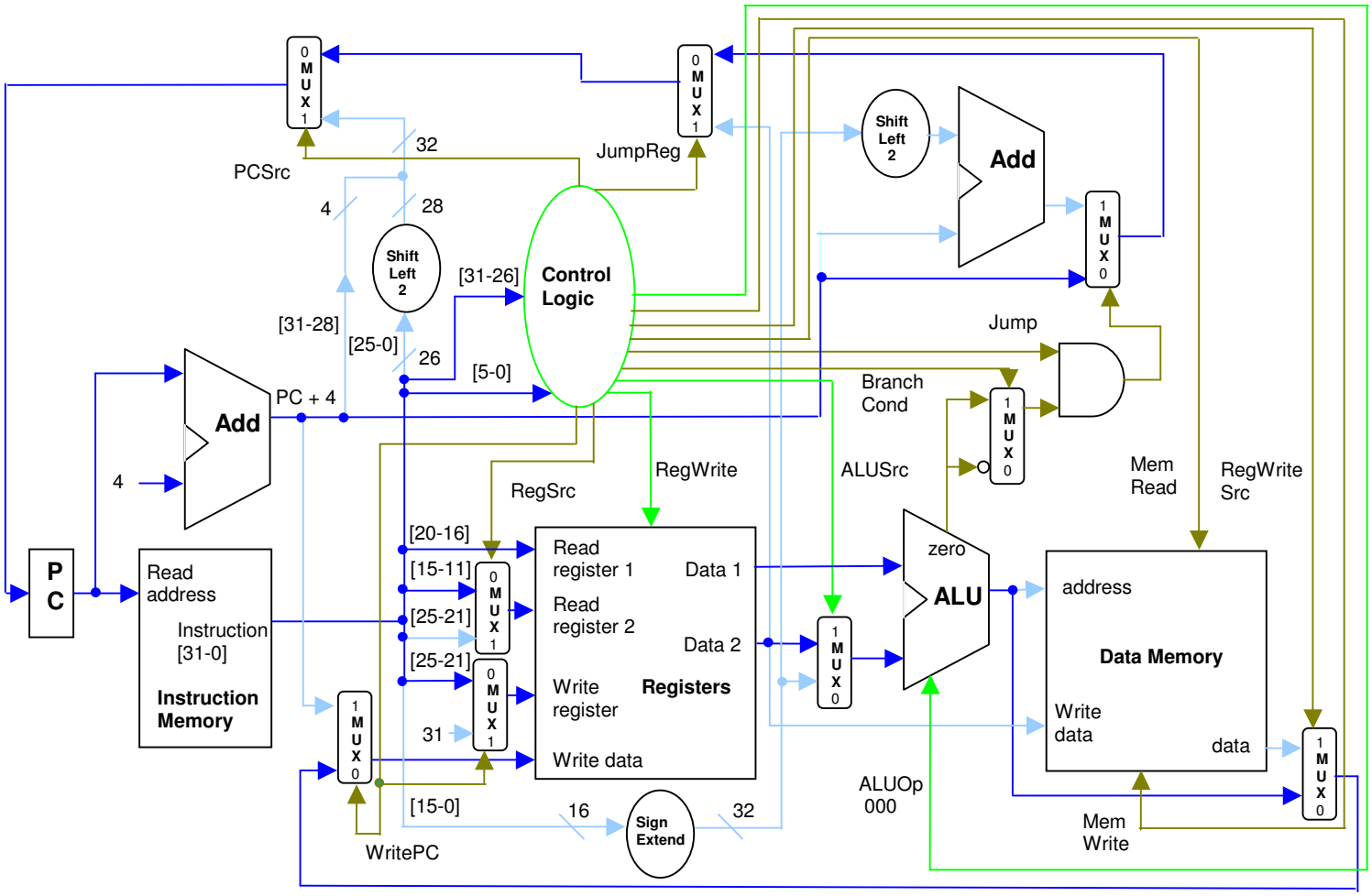
Befehl: add



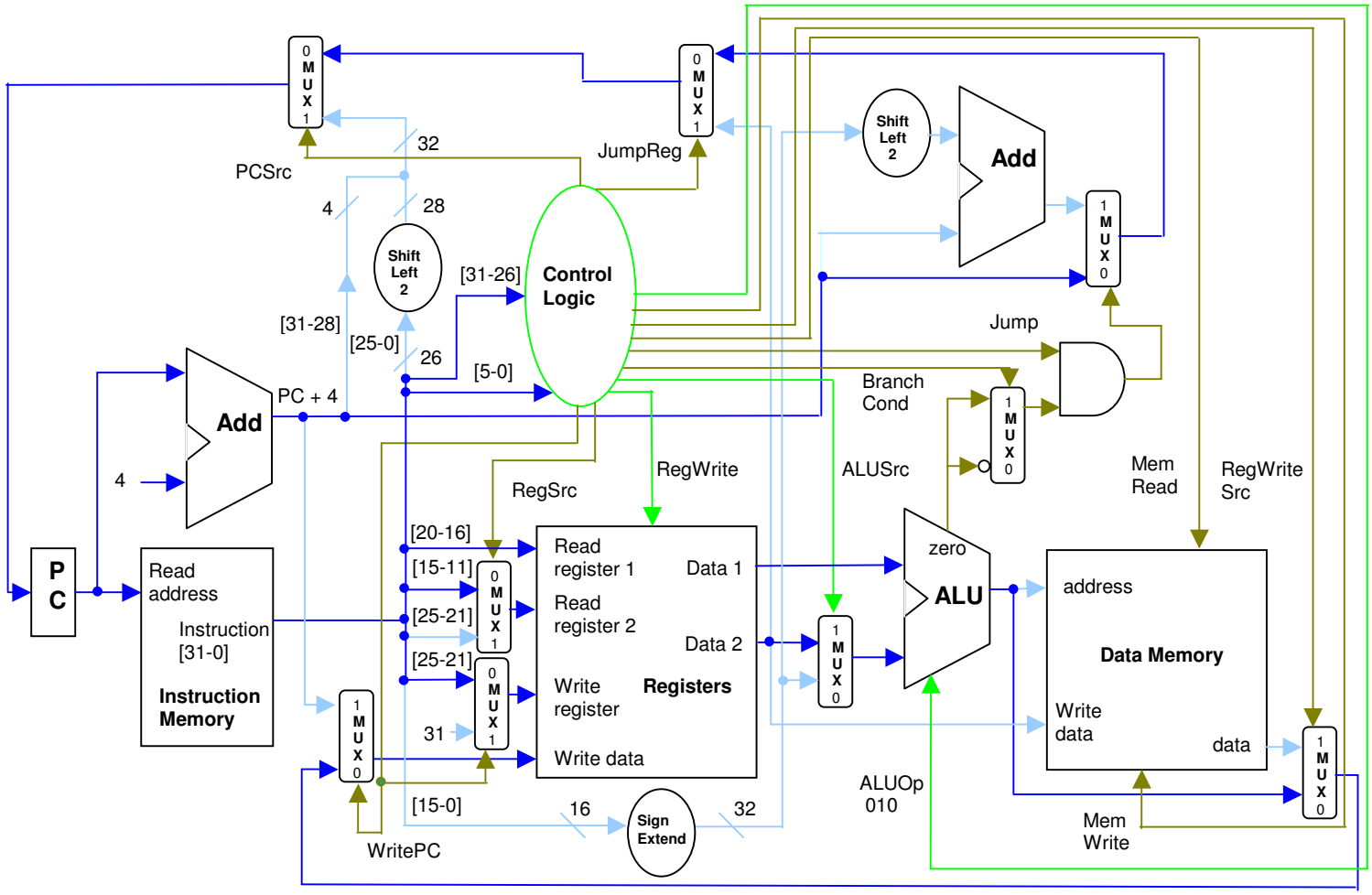
Befehl: sub


0 M U X 1
32
PCSrc
4
28
Shift Left 2
[31-28]
[25-0]
26
[31-26]
Control Logic
[5-0]
JumpReg
0 M U X 1
Shift Left 2
Add
1 M U X 0
Jump
Branch Cond
1 M U X 0
Add
PC + 4
4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
P C
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
0 M U X 1
0 M U X 1
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
1 M U X 0
31
[15-0]
16
Sign Extend
32
WritePC
1 M U X 0
zero
ALU
address
Data Memory
Write data
data
1 M U X 0
ALUOp 101
Mem Write

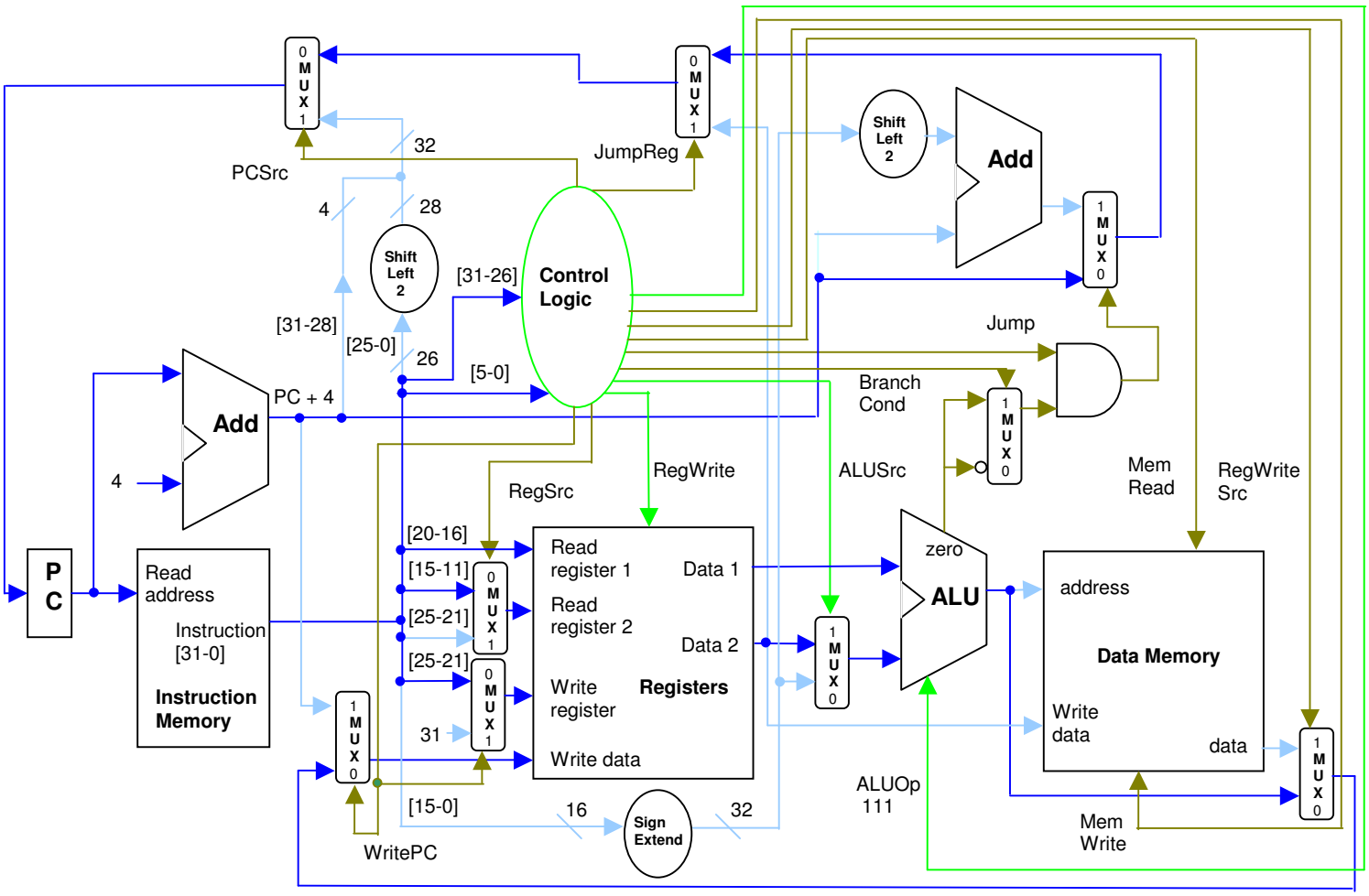
Befehl: and


0
M
U
X
1
PCSrc
32
4
28
Shift
Left
2
[31-28]
[25-0]
26
[31-26]
Control
Logic
JumpReg
Shift
Left
2
Add
1
M
U
X
0
Jump
Branch
Cond
[5-0]
Add
PC + 4
4
RegSrc
RegWrite
ALUSrc
Mem
Read
RegWrite
Src
P
C
Read
address
Instruction
[31-0]
Instruction
Memory
[20-16]
[15-11]
[25-21]
[25-21]
31
[15-0]
16
Sign
Extend
32
Read
register 1
Read
register 2
Write
register
Write data
Data 1
Data 2
Registers
zero
ALU
ALUOp
000
address
Data Memory
Write
data
data
Mem
Write
WritePC

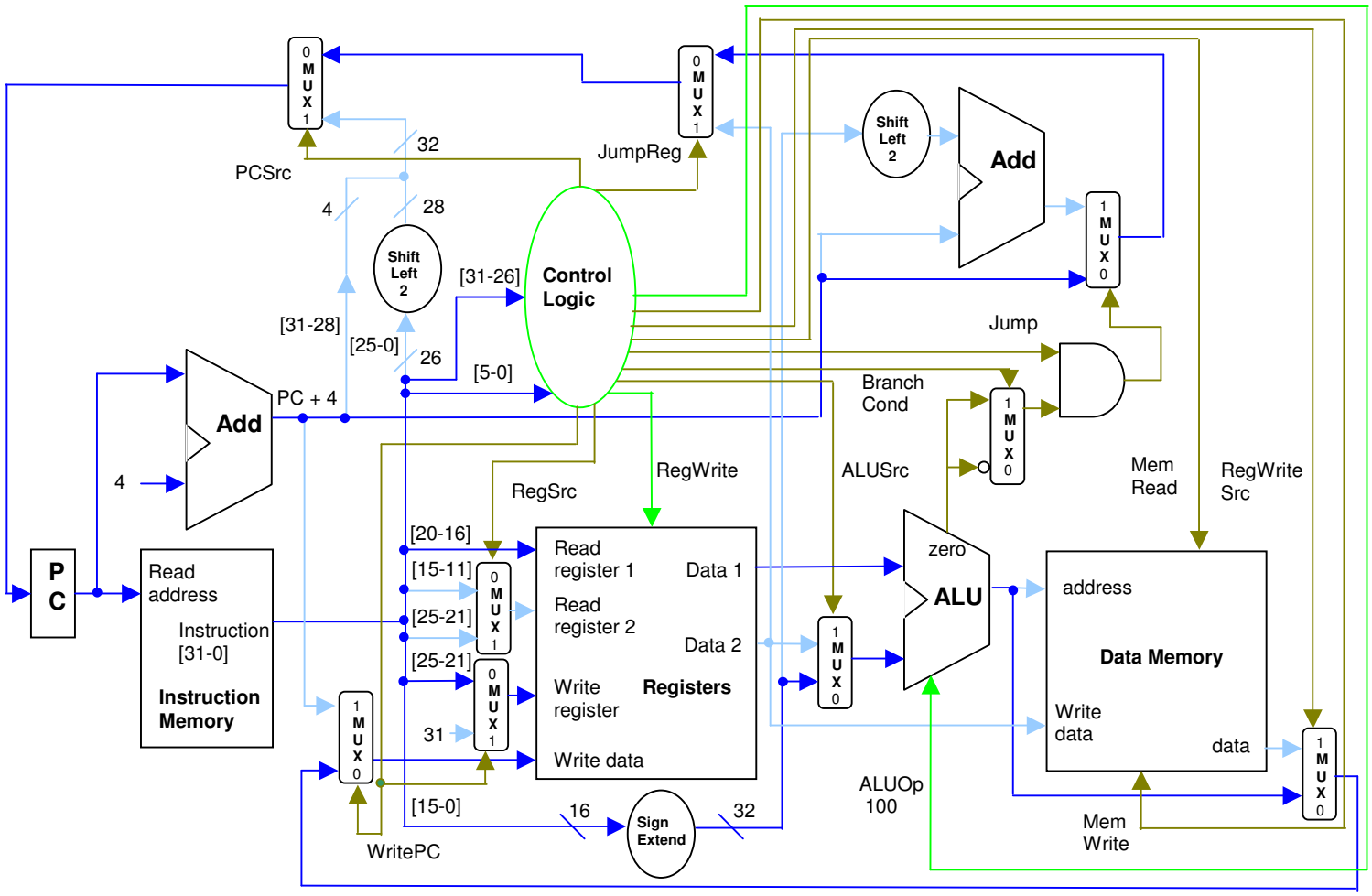
Befehl: or


0 MUX 1
32
PCSrc
4
28
Shift Left 2
[31-26]
Control Logic
[31-28]
[25-0]
26
[5-0]
JumpReg
0 MUX 1
Shift Left 2
Add
1 MUX 0
Jump
Branch Cond
1 MUX 0
Add
PC + 4
4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
P C
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
0 MUX 1
0 MUX 1
1 MUX 0
31
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
1 MUX 0
zero
ALU
address
Data Memory
Write data
data
1 MUX 0
ALUOp 010
Mem Write
[15-0]
16
Sign Extend
32
WritePC

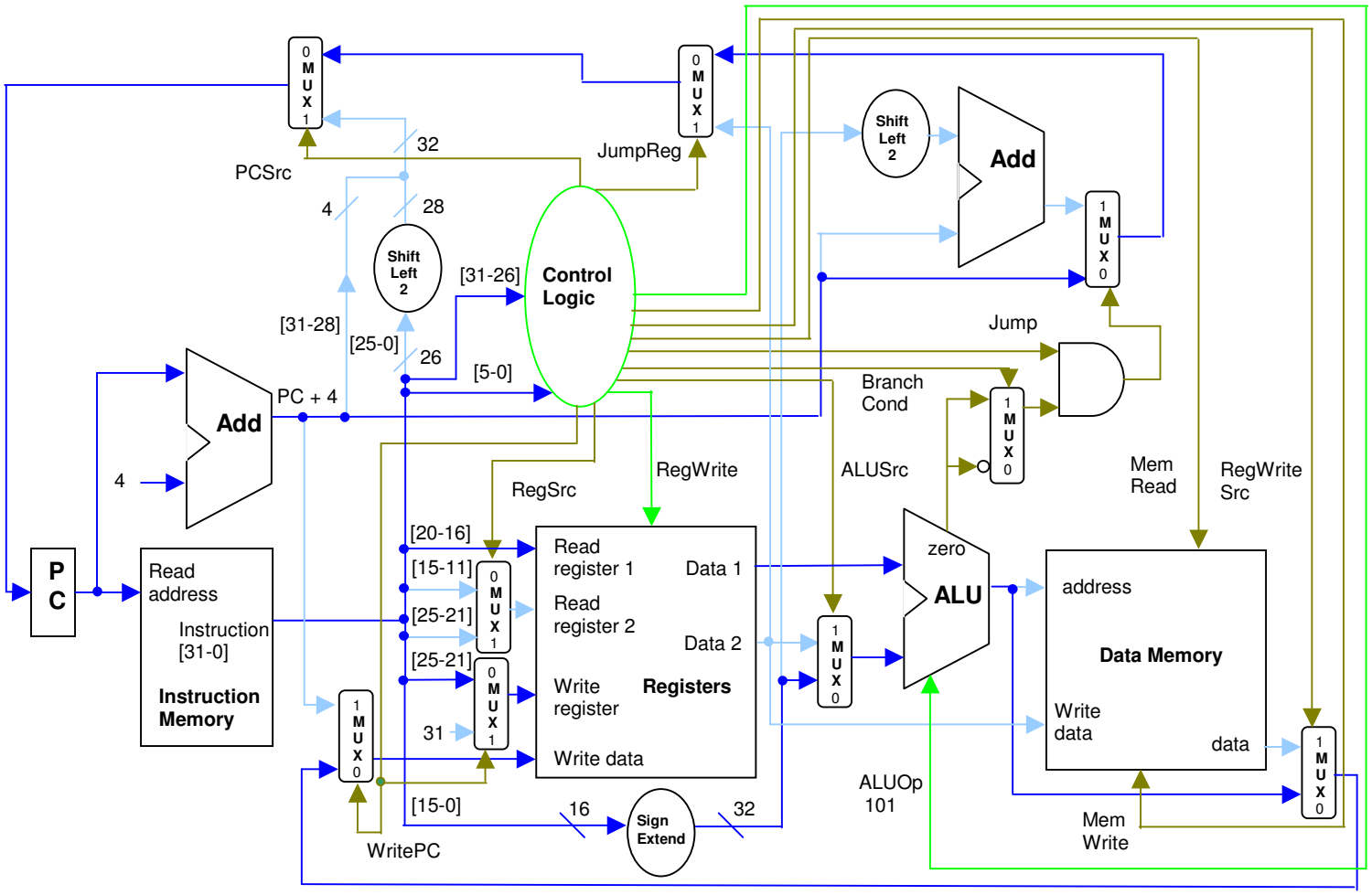
Befehl: slt


0 M U X 1
PCSrc
32
4
28
Shift Left 2
[31-28]
[25-0]
26
[31-26]
Control Logic
[5-0]
JumpReg
0 M U X 1
Shift Left 2
Add
1 M U X 0
Jump
Branch Cond
1 M U X 0
Add
PC + 4
4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
P C
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
0 M U X 1
0 M U X 1
1 M U X 0
31
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
1 M U X 0
zero
ALU
address
Data Memory
Write data
data
1 M U X 0
ALUOp 111
[15-0]
16
Sign Extend
32
WritePC
Mem Write

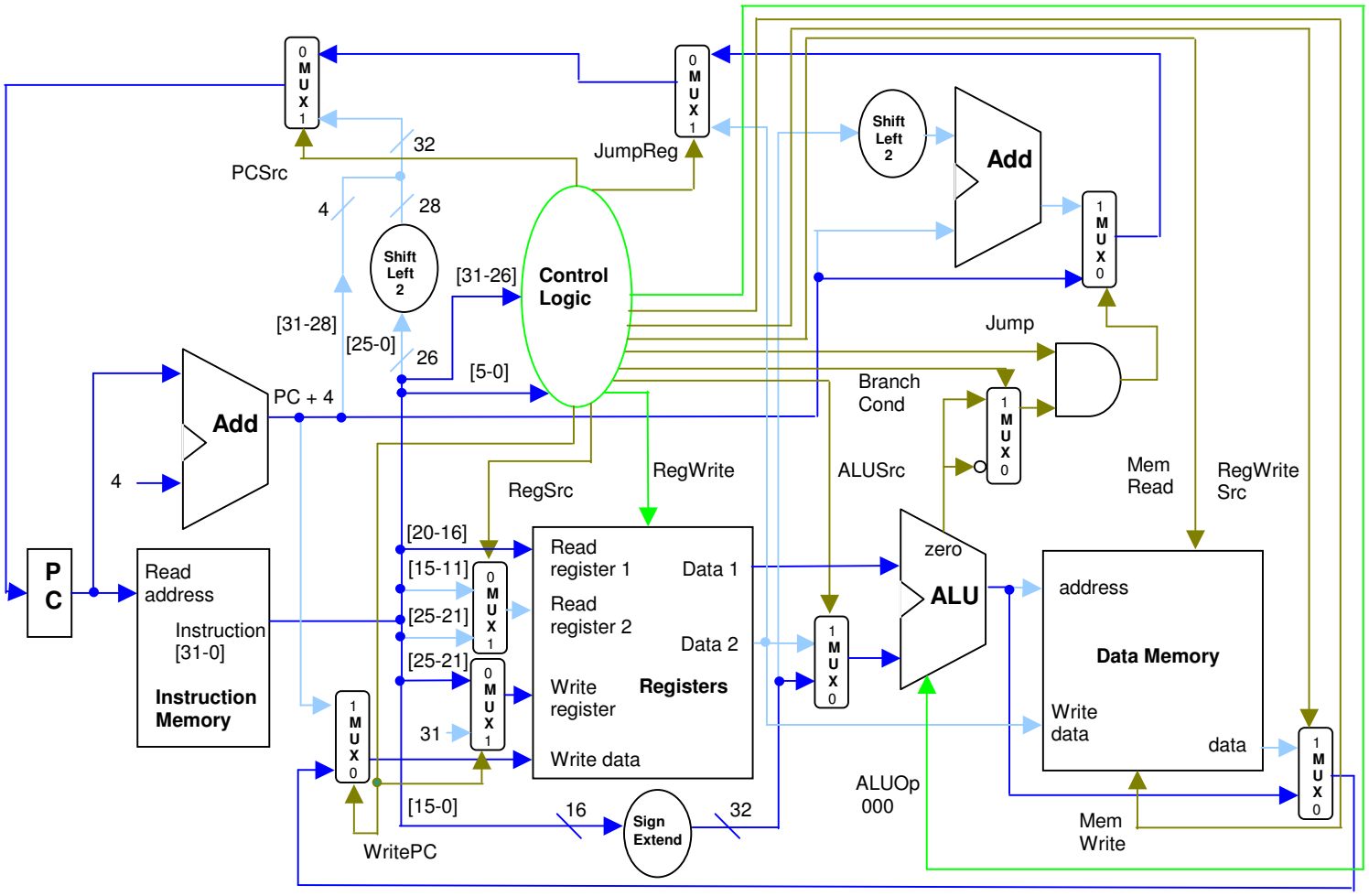
Befehl: addi


0
M
U
X
1
PCSrc
32
4
28
Shift
Left
2
[31-28]
[25-0]
26
[31-26]
Control
Logic
[5-0]
JumpReg
Shift
Left
2
Add
1
M
U
X
0
Jump
Branch
Cond
Add
PC + 4
4
RegSrc
RegWrite
ALUSrc
Mem
Read
RegWrite
Src
P
C
Read
address
Instruction
[31-0]
Instruction
Memory
[20-16]
[15-11]
[25-21]
[25-21]
31
Read
register 1
Read
register 2
Write
register
Write data
Data 1
Data 2
Registers
zero
ALU
address
Data Memory
Write
data
data
ALUOp
100
Mem
Write
[15-0]
16
Sign
Extend
32
WritePC

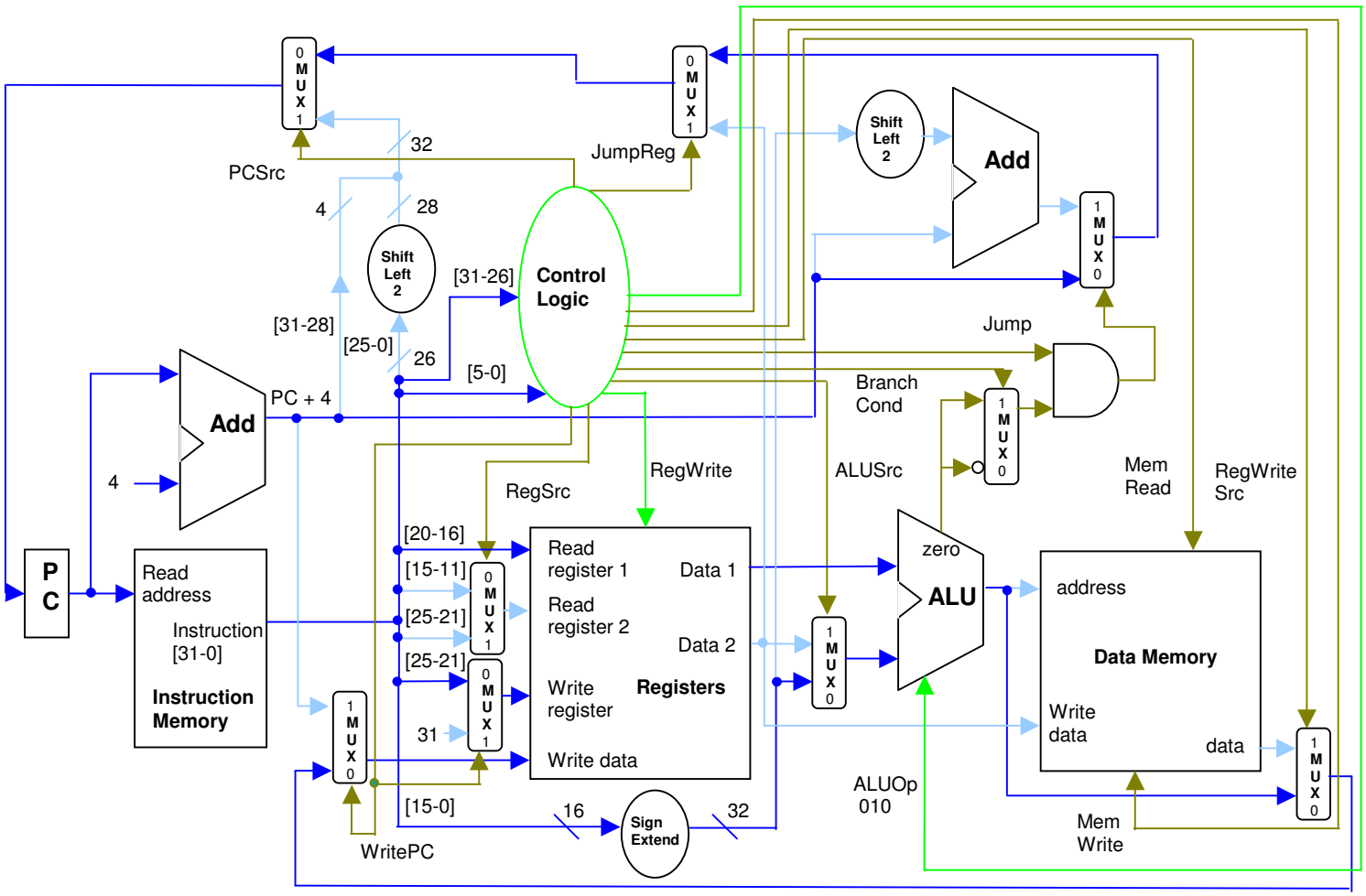
Befehl: subi


0 M U X 1
32
PCSrc
4
28
Shift Left 2
[31-28]
[25-0]
26
[31-26]
Control Logic
[5-0]
JumpReg
Shift Left 2
Add
1 M U X 0
Jump
Branch Cond
PC + 4
Add
4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
P C
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
31
[15-0]
16
Sign Extend
32
WritePC
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
zero
ALU
ALUOp 101
address
Data Memory
Write data
data
Mem Write

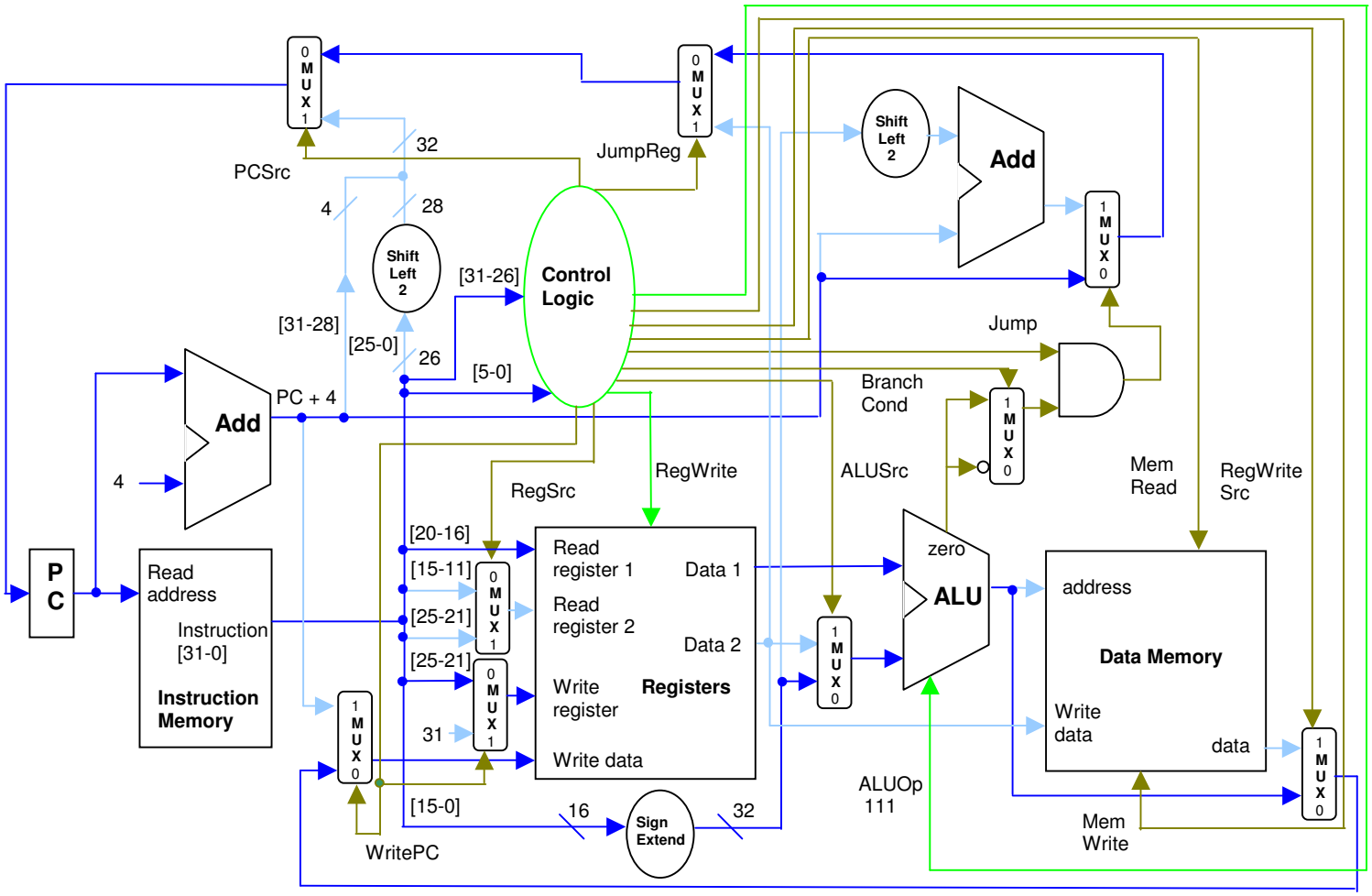
Befehl: andi


0 M U X 1
32
PCSrc
4
28
Shift Left 2
[31-28]
[25-0]
26
[31-26]
Control Logic
[5-0]
JumpReg
0 M U X 1
Shift Left 2
Add
1 M U X 0
Jump
Branch Cond
1 M U X 0
Add
PC + 4
4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
P C
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
0 M U X 1
0 M U X 1
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
1 M U X 0
31
[15-0]
16
Sign Extend
32
WritePC
zero
ALU
address
Data Memory
Write data
data
1 M U X 0
ALUOp 000
Mem Write

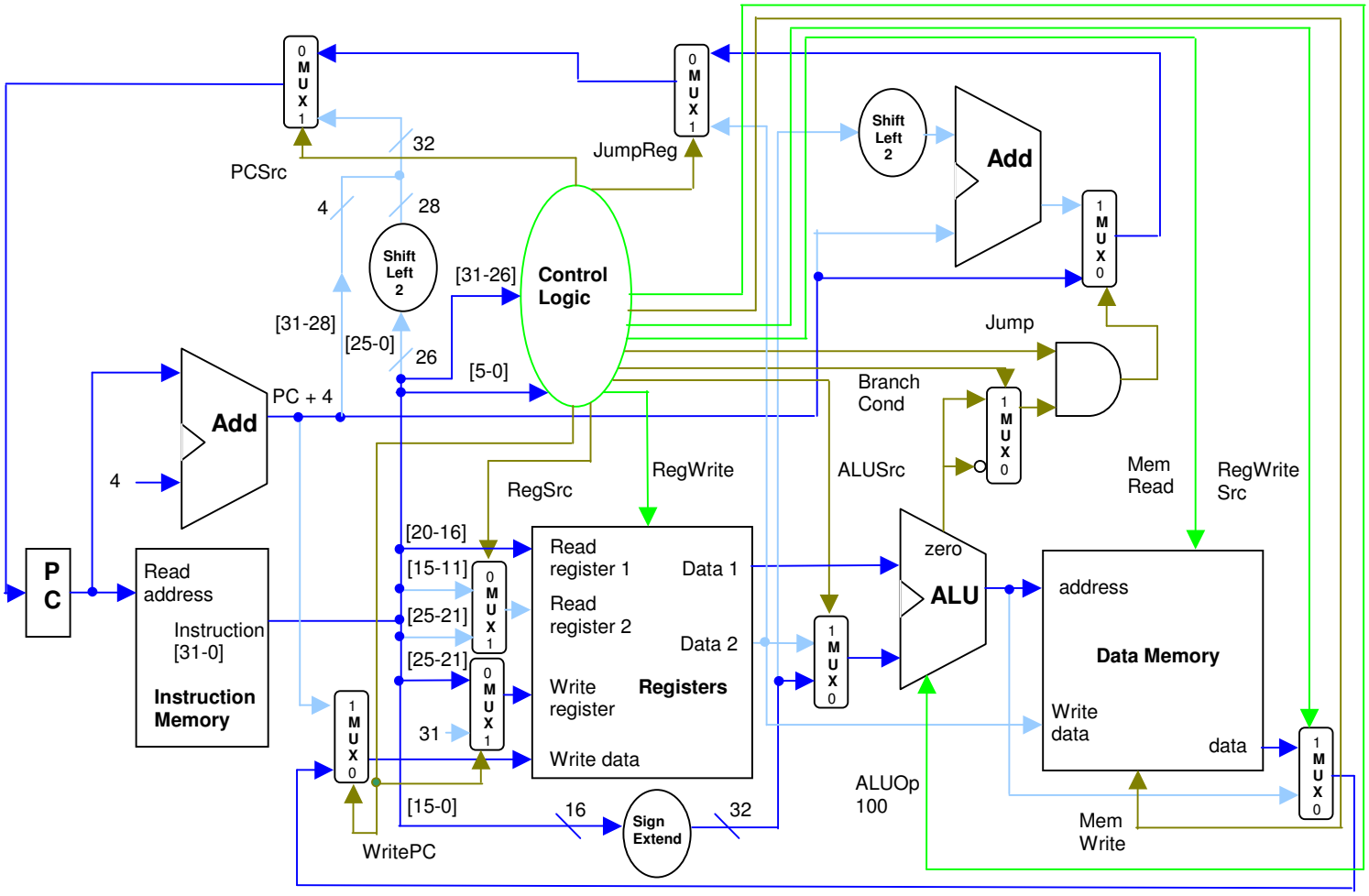
Befehl: ori


0 M U X 1
PCSrc
32
4
28
Shift Left 2
[31-28]
[25-0]
26
[31-26]
Control Logic
[5-0]
JumpReg
0 M U X 1
Shift Left 2
Add
1 M U X 0
Jump
Branch Cond
1 M U X 0
Add
PC + 4
4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
P C
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
0 M U X 1
0 M U X 1
1 M U X 0
31
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
1 M U X 0
zero
ALU
address
Data Memory
Write data
data
1 M U X 0
ALUOp 010
Mem Write
[15-0]
16
Sign Extend
32
WritePC

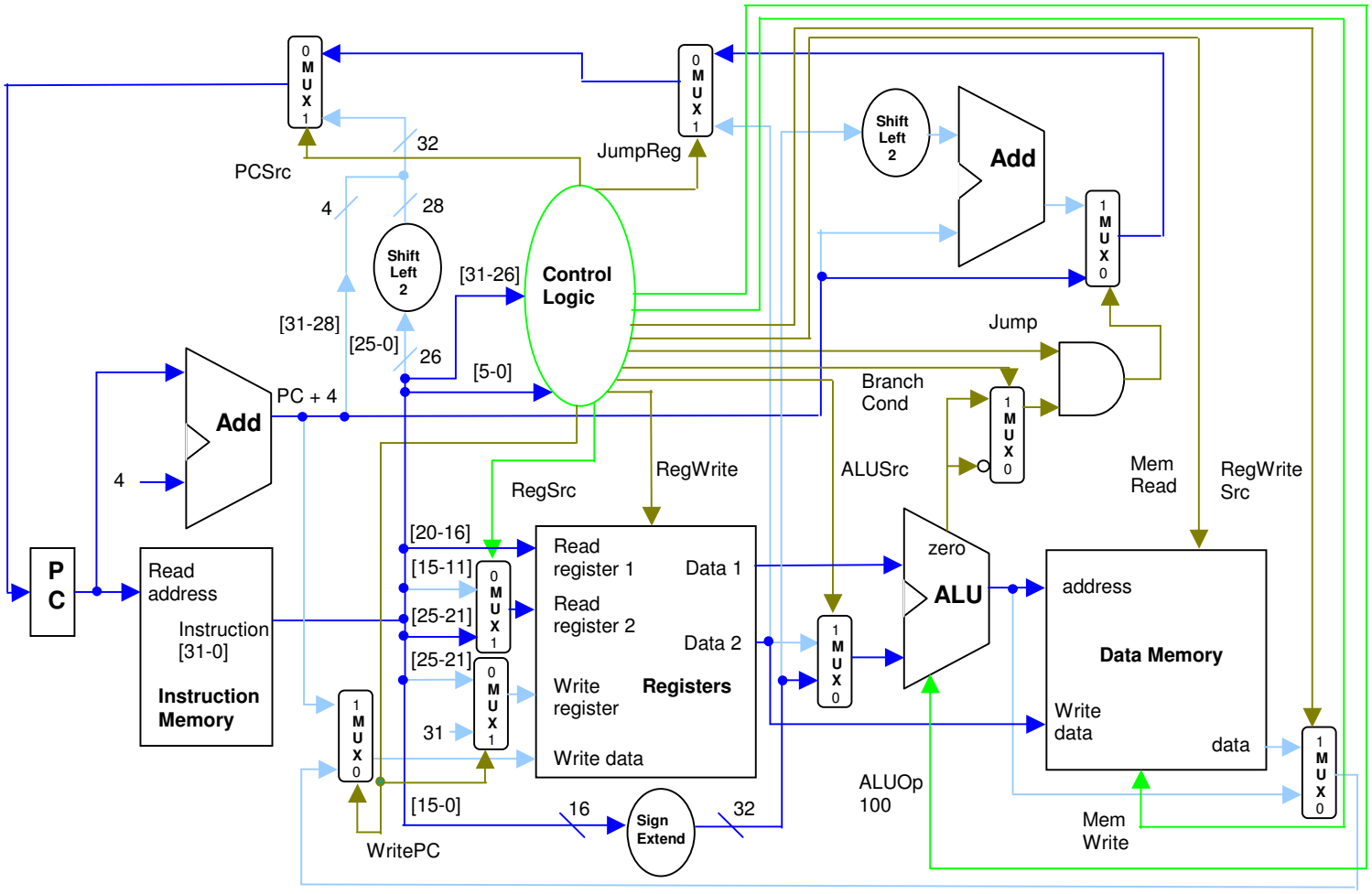
Befehl: slti


0 M U X 1
32
PCSrc
4
28
Shift Left 2
[31-26]
Control Logic
JumpReg
Shift Left 2
Add
1 M U X 0
[31-28]
[25-0]
26
[5-0]
Jump
Branch Cond
Add
PC + 4
4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
[20-16]
[15-11]
[25-21]
[25-21]
31
[15-0]
16
Sign Extend
32
zero
ALU
P C
Read address
Instruction [31-0]
Instruction Memory
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
address
Data Memory
Write data
data
ALUOp 111
Mem Write
WritePC

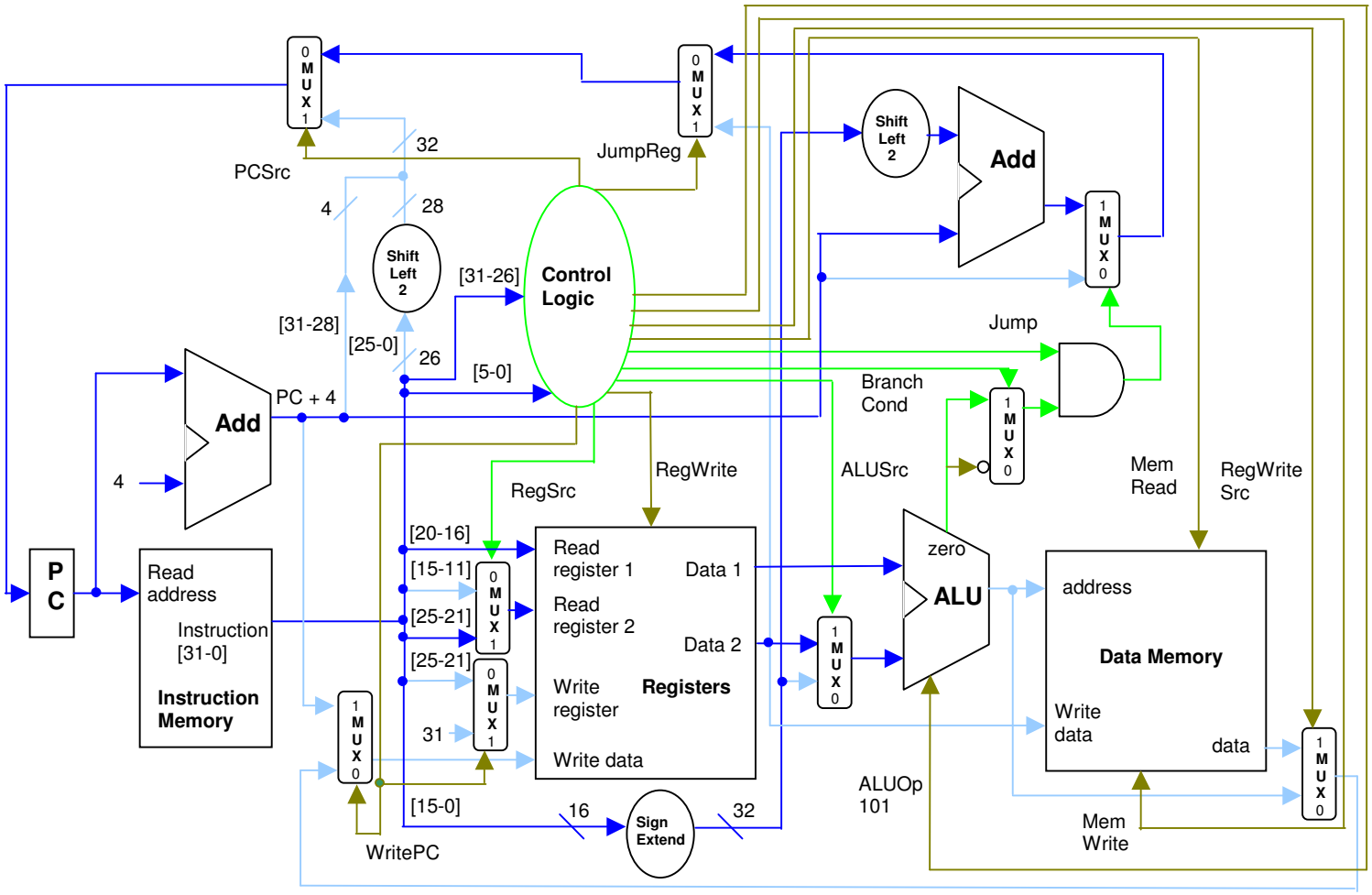
Befehl: lw


0 MUX 1
32
PCSrc
4
28
Shift Left 2
[31-28]
[25-0]
26
[31-26]
Control Logic
[5-0]
JumpReg
0 MUX 1
Shift Left 2
Add
1 MUX 0
Jump
Branch Cond
1 MUX 0
Add
PC + 4
4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
PC
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
0 MUX 1
0 MUX 1
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
1 MUX 0
31
[15-0]
16
Sign Extend
32
WritePC
zero
ALU
1 MUX 0
address
Data Memory
Write data
data
1 MUX 0
ALUOp
100
Mem Write

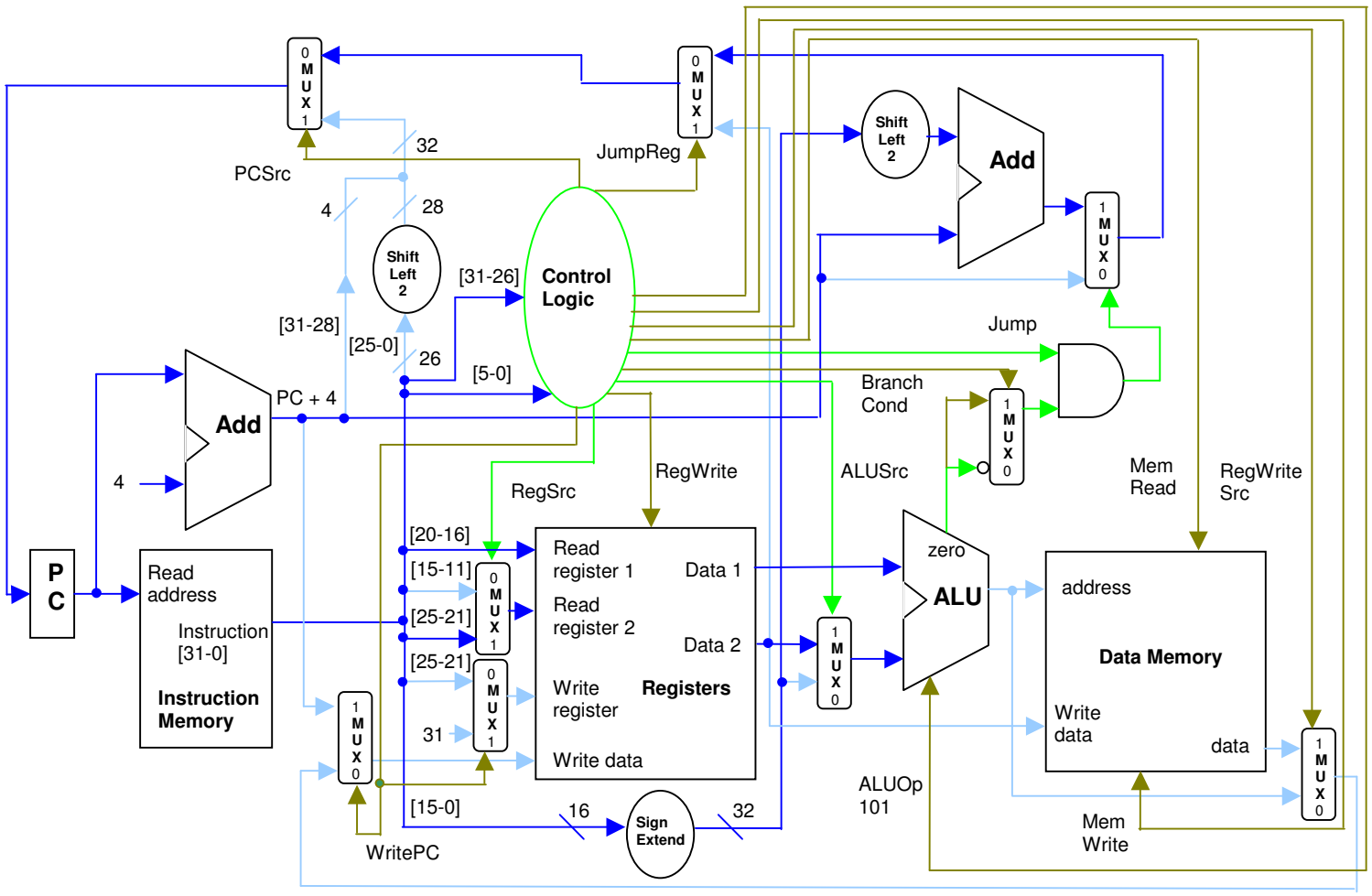
Befehl: sw


0 M U X 1
32
PCSrc
JumpReg
4
28
Shift Left 2
[31-26]
Control Logic
[31-28]
[25-0]
26
[5-0]
Add
PC + 4
4
RegSrc
RegWrite
[20-16]
[15-11]
[25-21]
[25-21]
31
[15-0]
16
Sign Extend
32
WritePC
P C
Read address
Instruction [31-0]
Instruction Memory
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
Shift Left 2
Add
Jump
Branch Cond
ALUSrc
zero
ALU
ALUOp 100
Mem Read
RegWrite Src
address
Data Memory
Write data
data
Mem Write

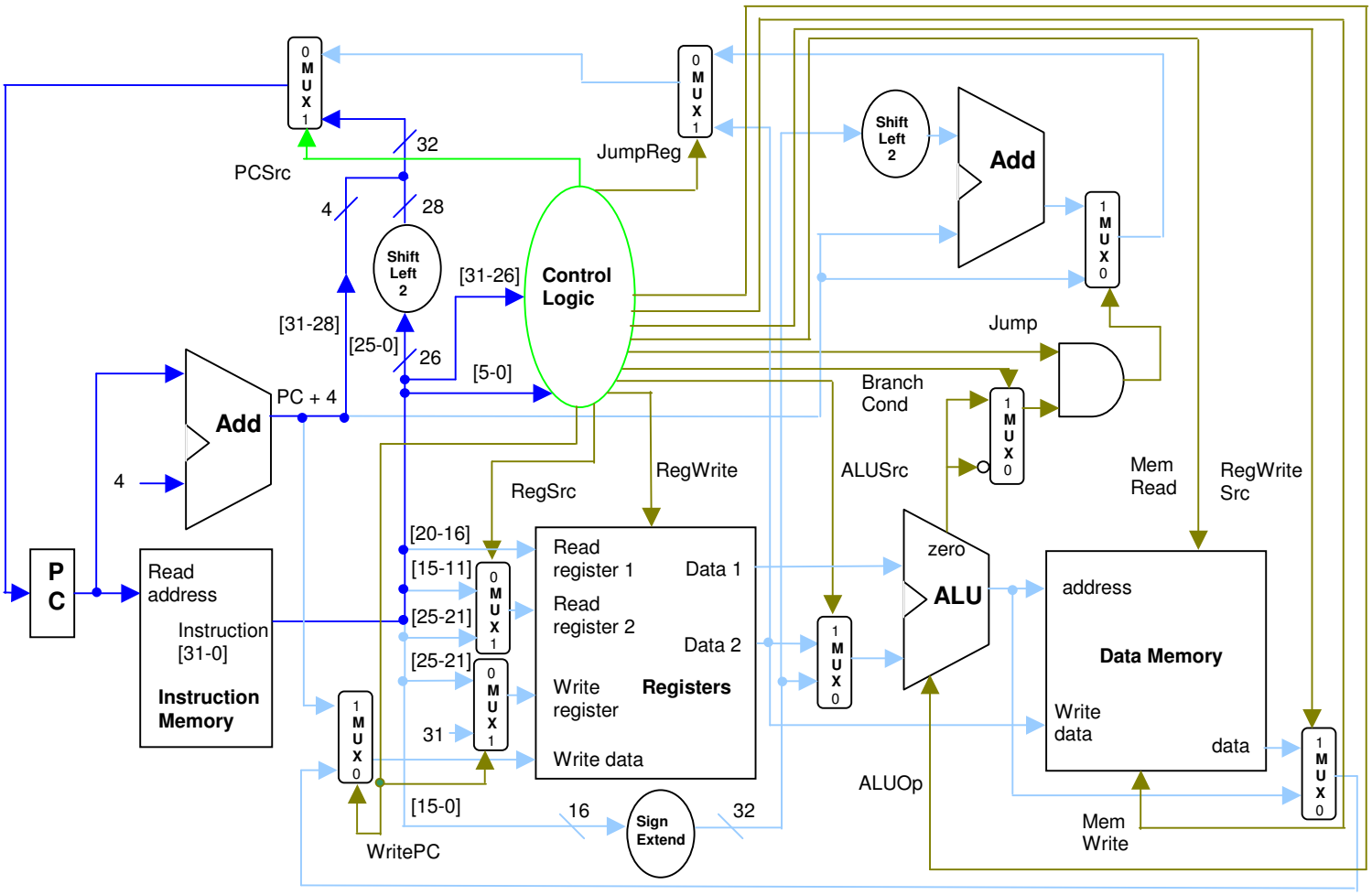
Befehl: beq


0 M U X 1
PCSrc
32
4
28
Shift Left 2
[31-26]
Control Logic
[31-28]
[25-0]
26
[5-0]
JumpReg
Shift Left 2
Add
1 M U X 0
Jump
Branch Cond
PC + 4
Add
4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
P C
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
31
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
zero
ALU
address
Data Memory
Write data
data
ALUOp 101
Mem Write
[15-0]
16
Sign Extend
32
WritePC

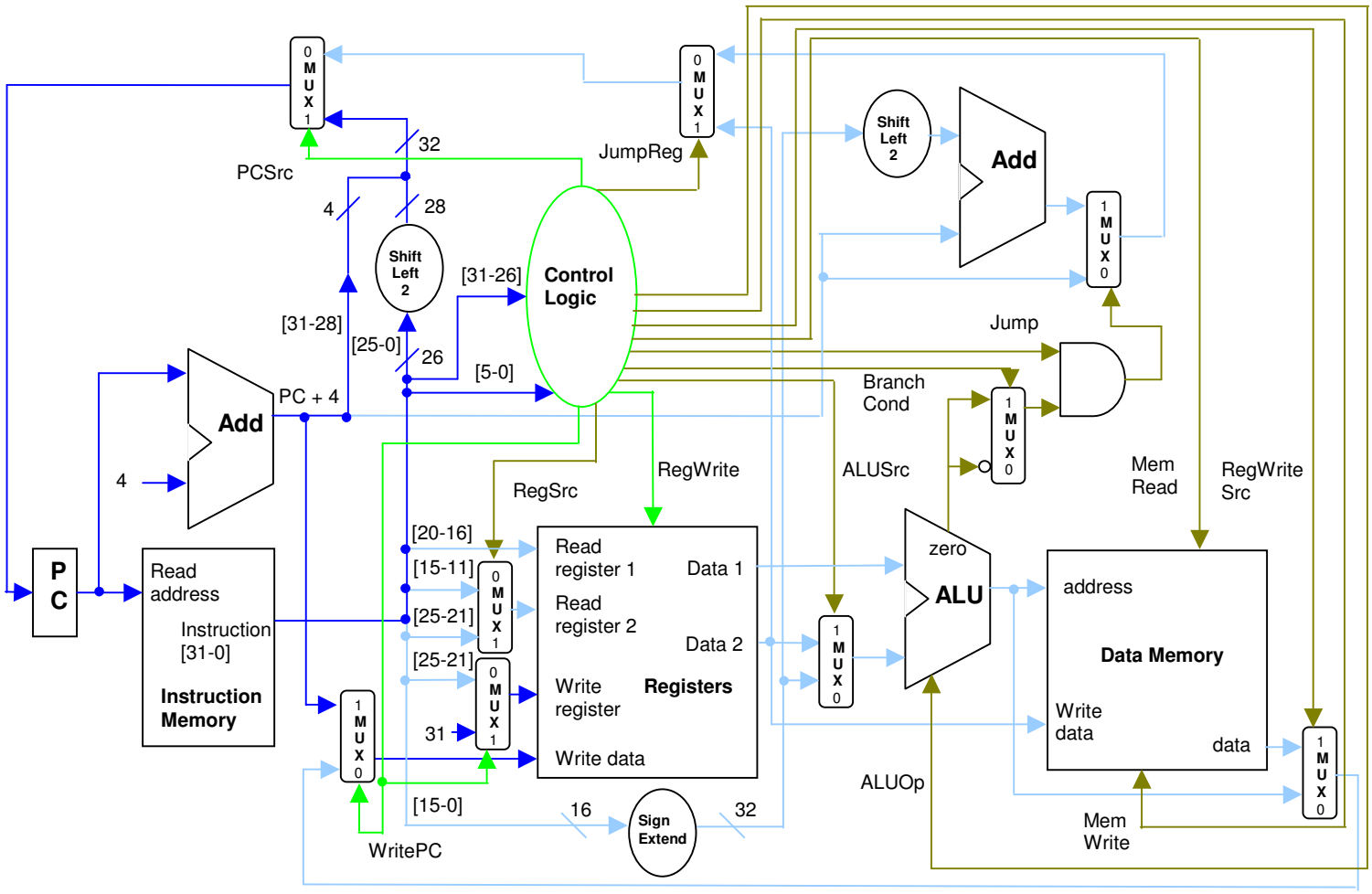
Befehl: bne


0 MUX 1
32
PCSrc
4
28
Shift Left 2
[31-28]
[25-0]
26
[31-26]
Control Logic
[5-0]
JumpReg
0 MUX 1
Shift Left 2
Add
1 MUX 0
Jump
Branch Cond
1 MUX 0
Add
4
PC + 4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
P C
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
0 MUX 1
0 MUX 1
1 MUX 0
31
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
zero
ALU
1 MUX 0
address
Data Memory
Write data
data
1 MUX 0
ALUOp 101
Mem Write
[15-0]
16
Sign Extend
32
WritePC

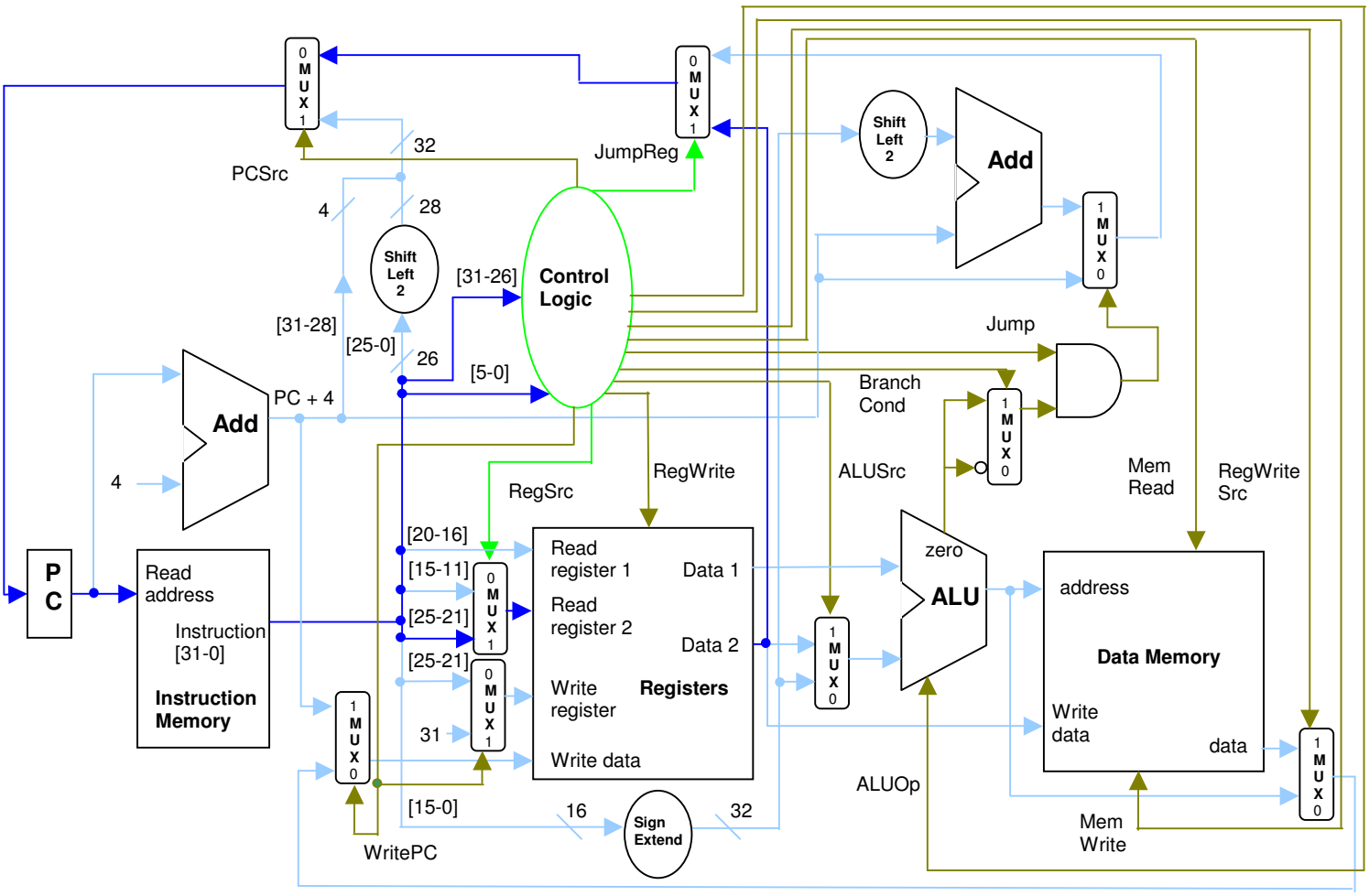
Befehl: j


0 MUX 1
PCSrc
32
4
28
Shift Left 2
[31-28]
[25-0]
26
[31-26]
Control Logic
[5-0]
JumpReg
0 MUX 1
Shift Left 2
Add
1 MUX 0
Jump
Branch Cond
1 MUX 0
Add
PC + 4
4
RegSrc
RegWrite
ALUSrc
Mem Read
RegWrite Src
PC
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
0 MUX 1
0 MUX 1
1 MUX 0
31
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
1 MUX 0
zero
ALU
address
Data Memory
Write data
data
1 MUX 0
ALUOp
Mem Write
[15-0]
16
Sign Extend
32
WritePC

Befehl: jal

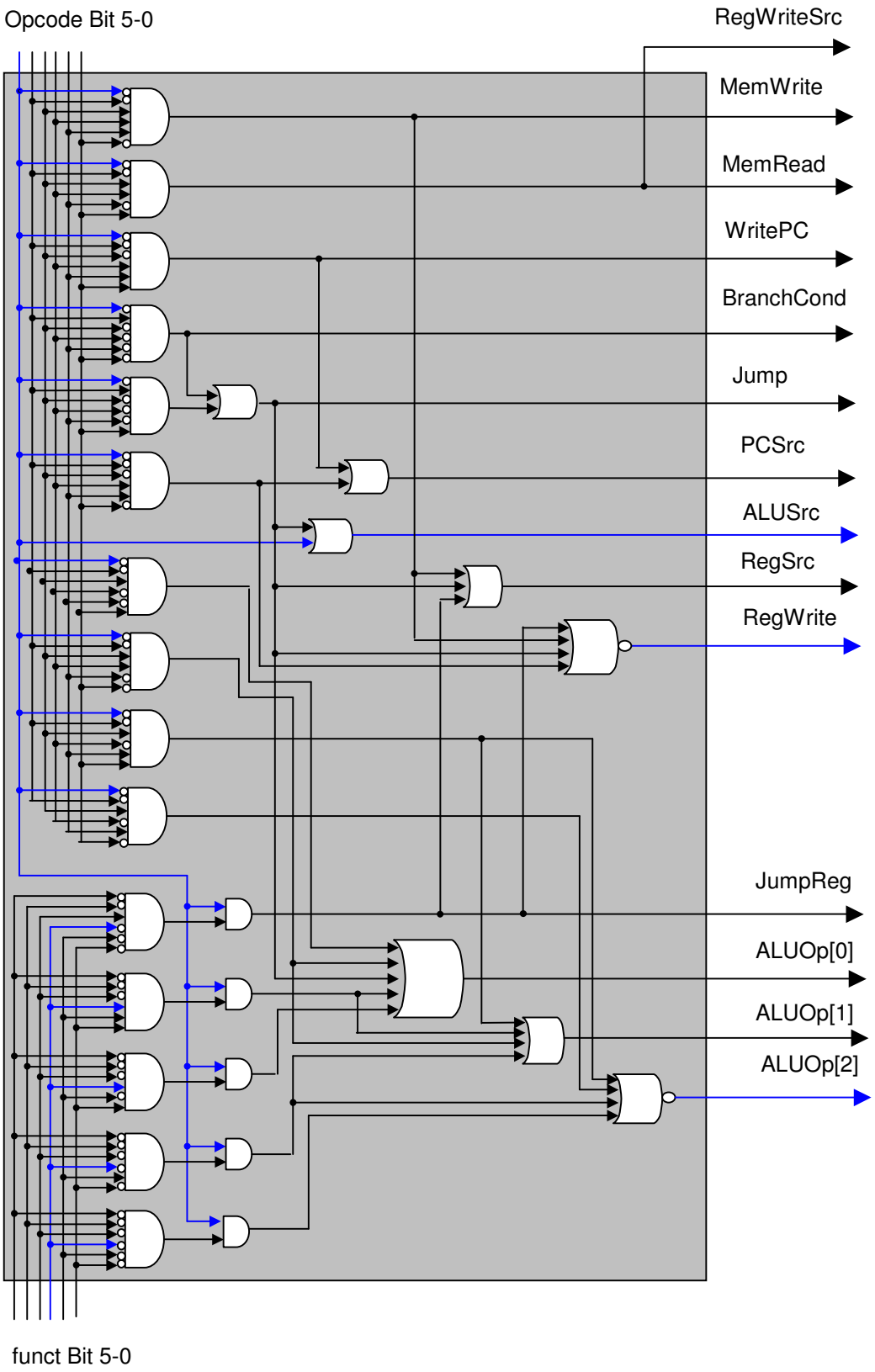

0 M U X 1
PCSrc
32
4
28
Shift Left 2
[31-28]
[25-0]
26
[31-26]
Control Logic
[5-0]
JumpReg
Shift Left 2
Add
1 M U X 0
Jump
Branch Cond
ALUSrc
Mem Read
RegWrite Src
RegWrite
RegSrc
PC + 4
Add
4
P C
Read address
Instruction [31-0]
Instruction Memory
[20-16]
[15-11]
[25-21]
[25-21]
31
[15-0]
16
Sign Extend
32
Read register 1
Read register 2
Write register
Write data
Data 1
Data 2
Registers
zero
ALU
address
Data Memory
Write data
data
ALUOp
Mem Write
WritePC

Befehl: jr

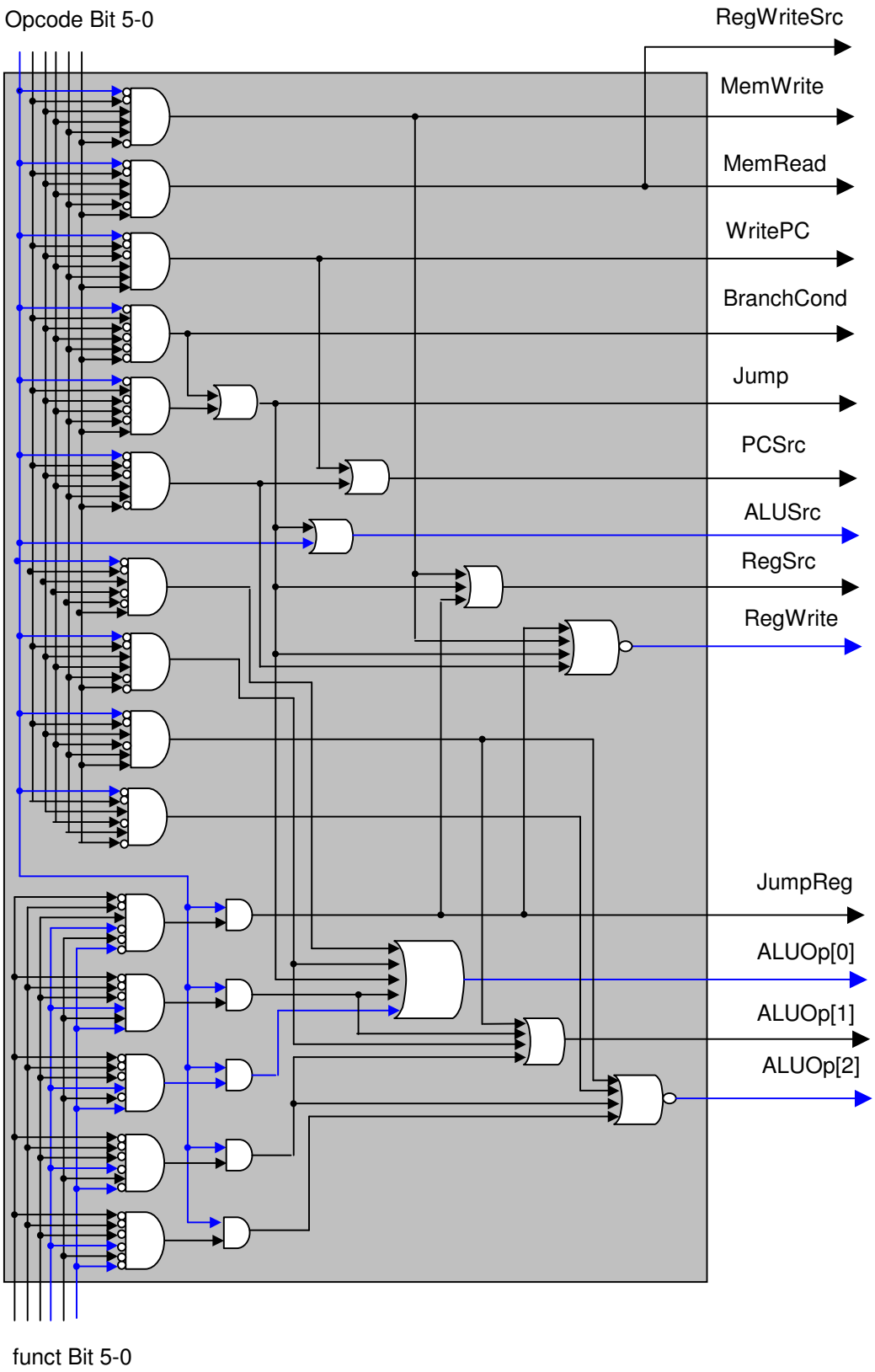

0
M
U
X
1
PCSrc
32
JumpReg
4
28
Shift
Left
2
[31-26]
Control
Logic
[31-28]
[25-0]
26
[5-0]
Add
PC + 4
4
RegSrc
RegWrite
[20-16]
[15-11]
[25-21]
[25-21]
31
[15-0]
16
Sign
Extend
32
WritePC
P
C
Read
address
Instruction
[31-0]
Instruction
Memory
Read
register 1
Read
register 2
Write
register
Write data
Data 1
Data 2
Registers
Shift
Left
2
Add
Jump
Branch
Cond
ALUSrc
zero
ALU
ALUOp
Mem
Read
RegWrite
Src
address
Data Memory
Write
data
data
Mem
Write

Aufgabe 5.4

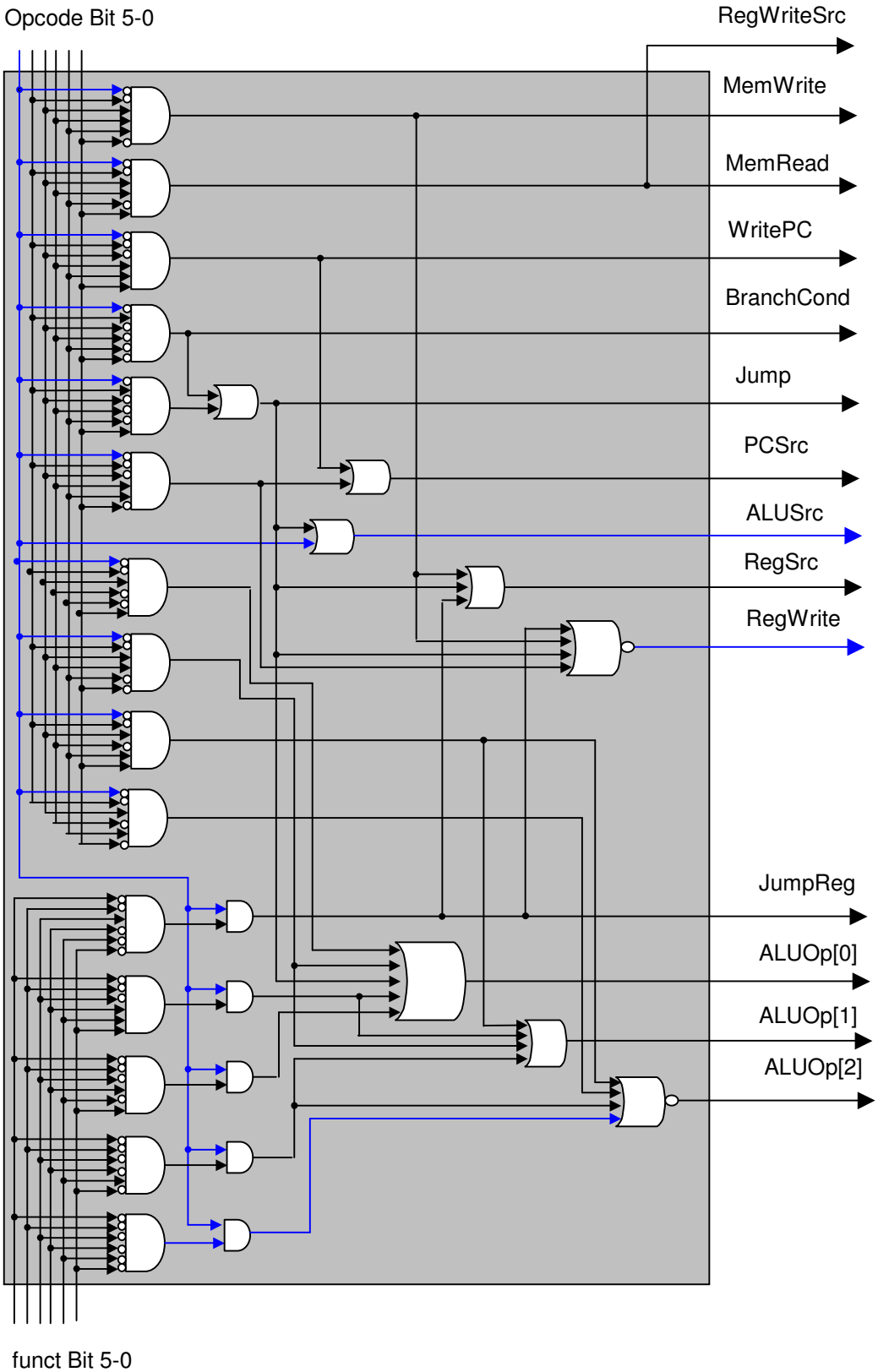
Befehl: add
Opcode: 100000
funct: 000100



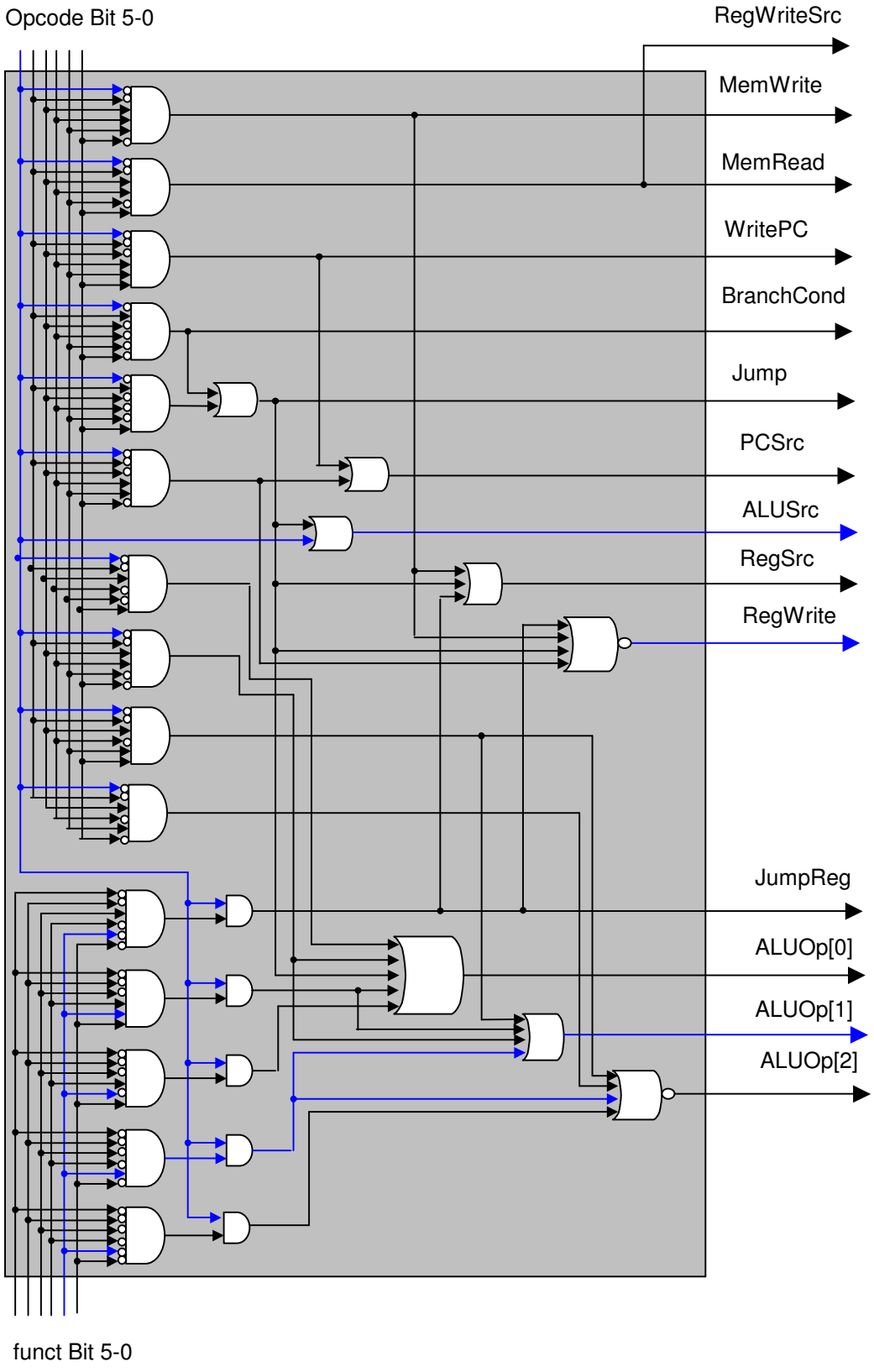
Befehl: sub
Opcode: 100000
funct: 000101



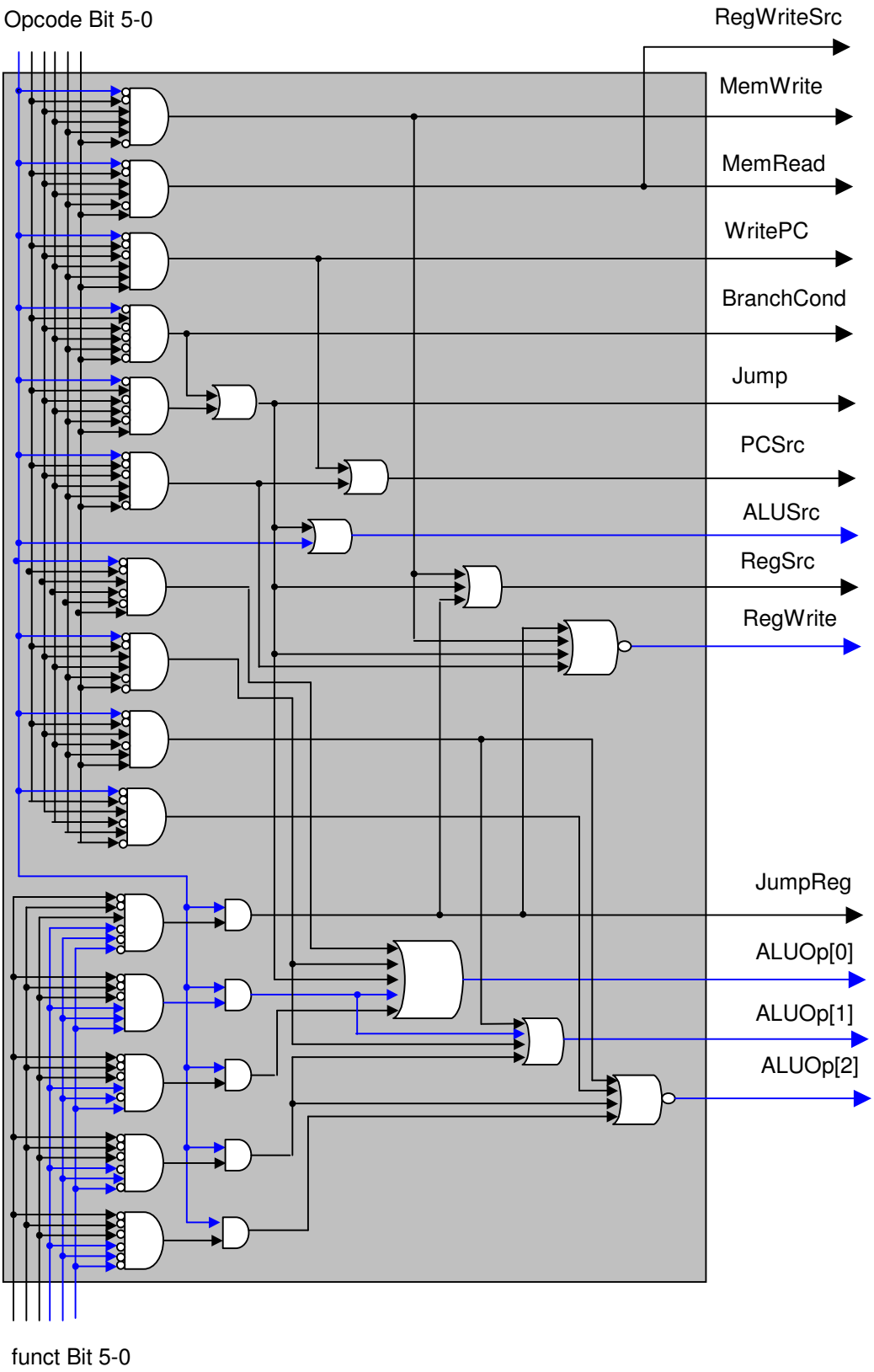
Befehl: and
Opcode: 100000
funct: 000000



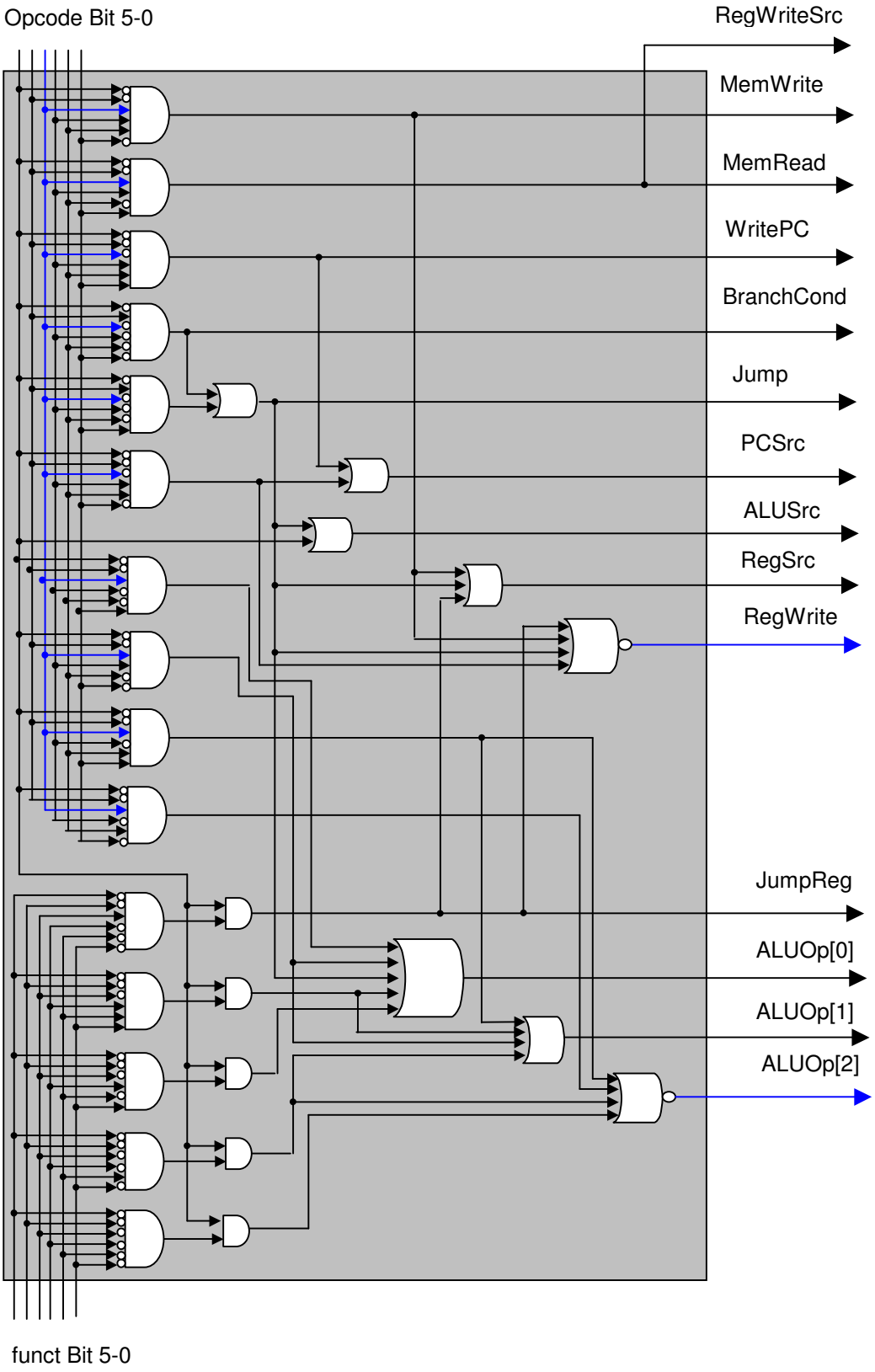
Befehl: or
Opcode: 100000
funct: 000020



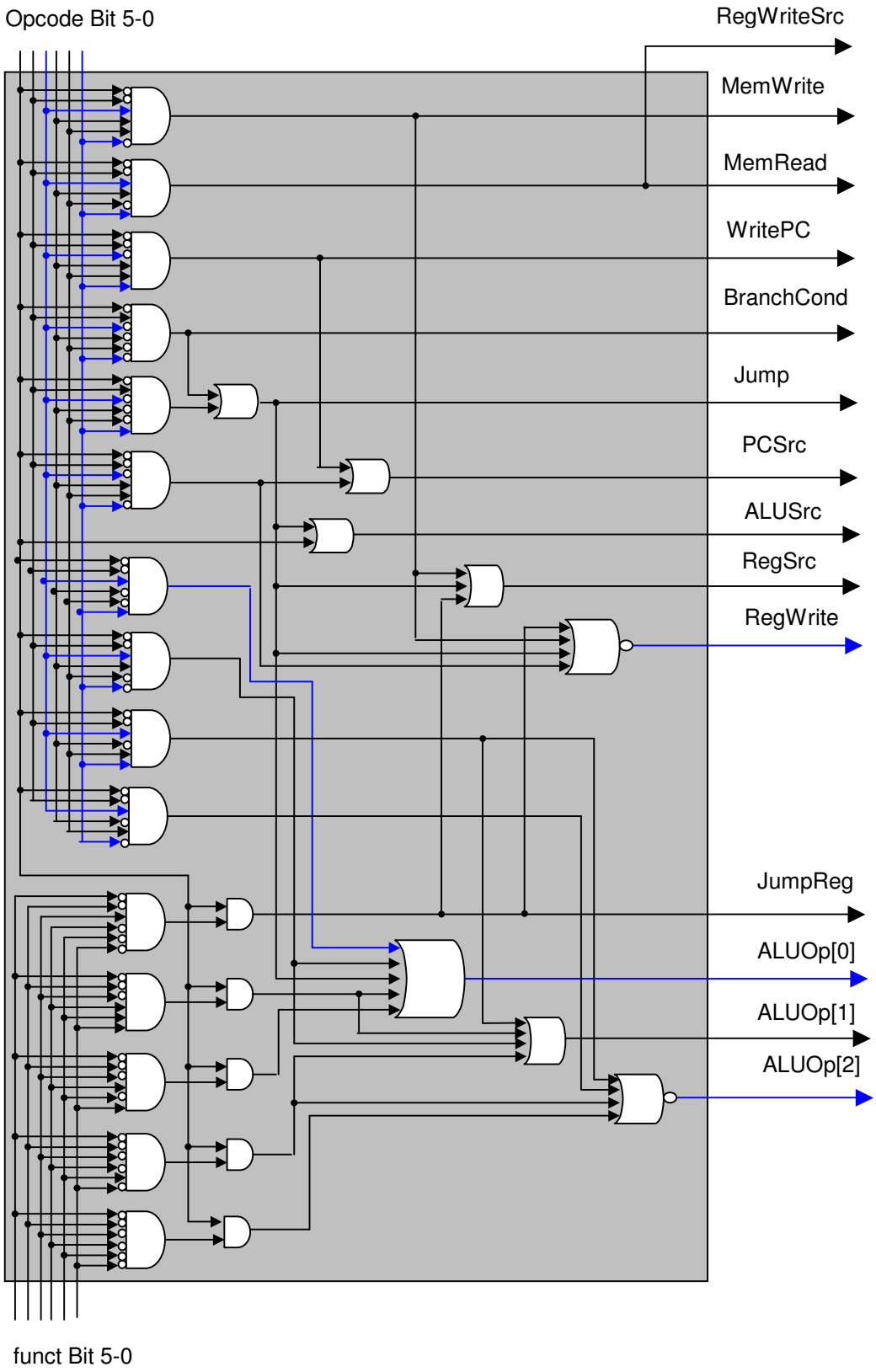
Befehl: slt
Opcode: 100000
funct: 000111



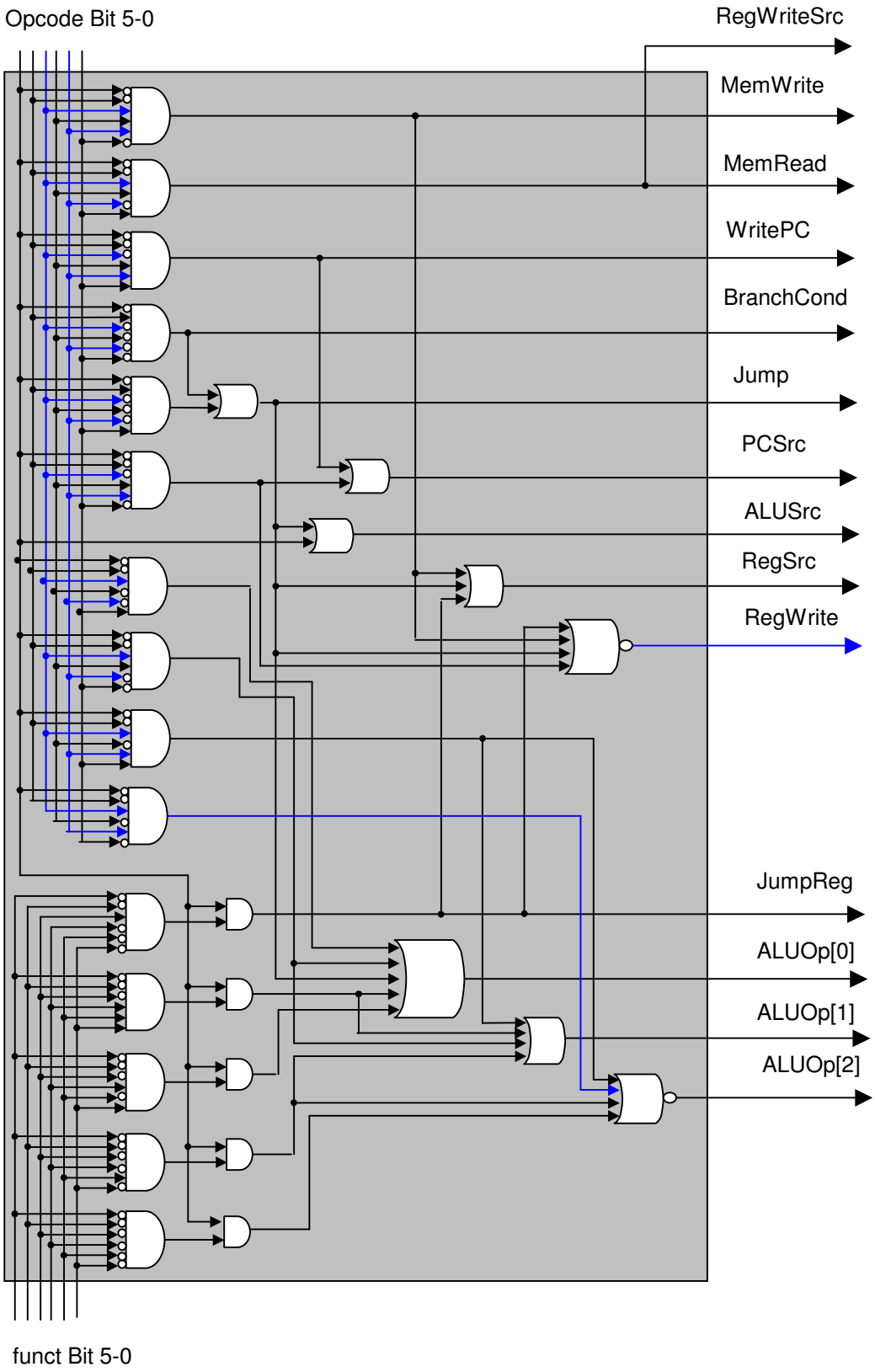
Befehl: addi
Opcode: 001000
funct: -



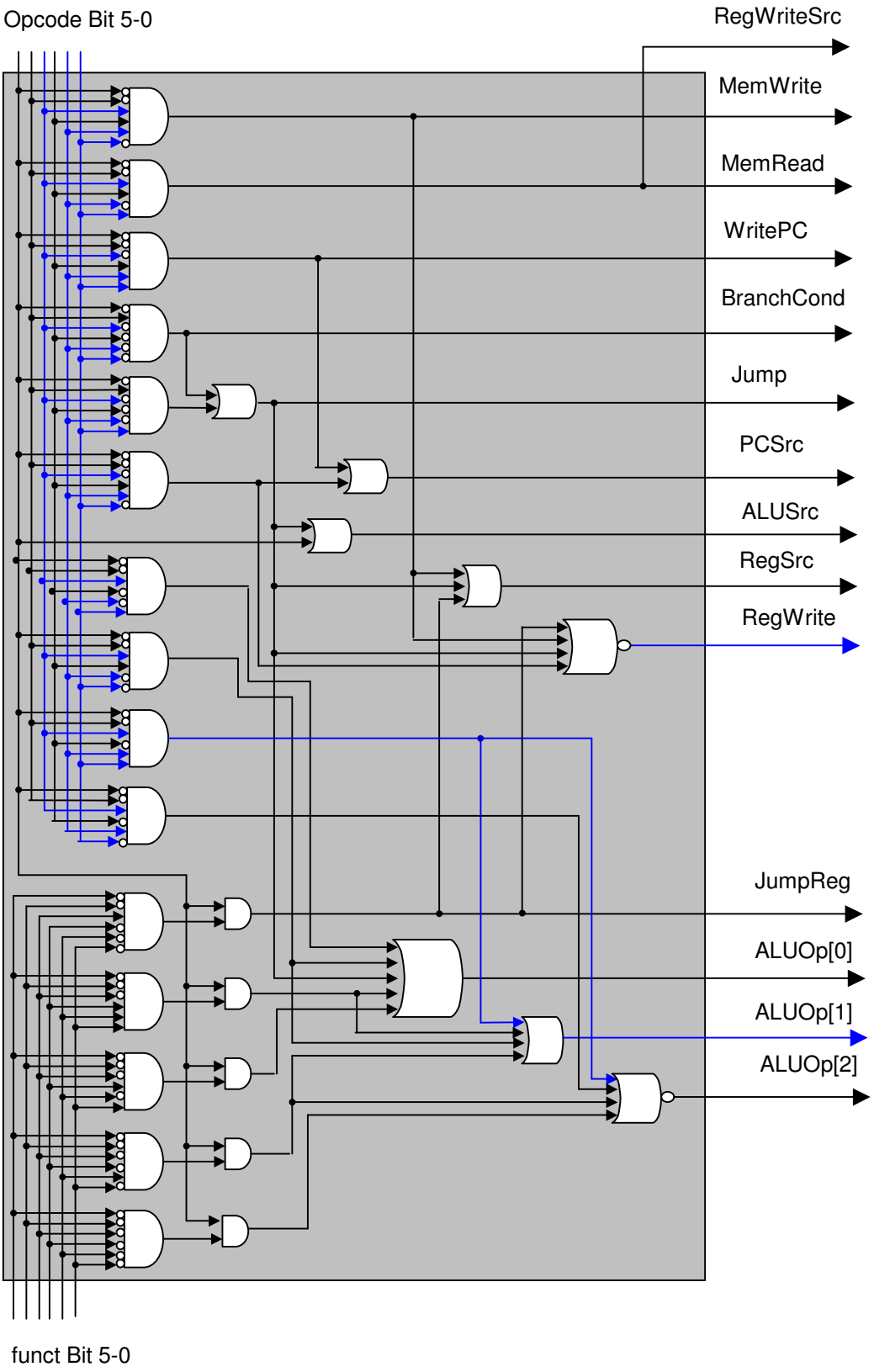
Befehl: subi
Opcode: 001001
funct: -



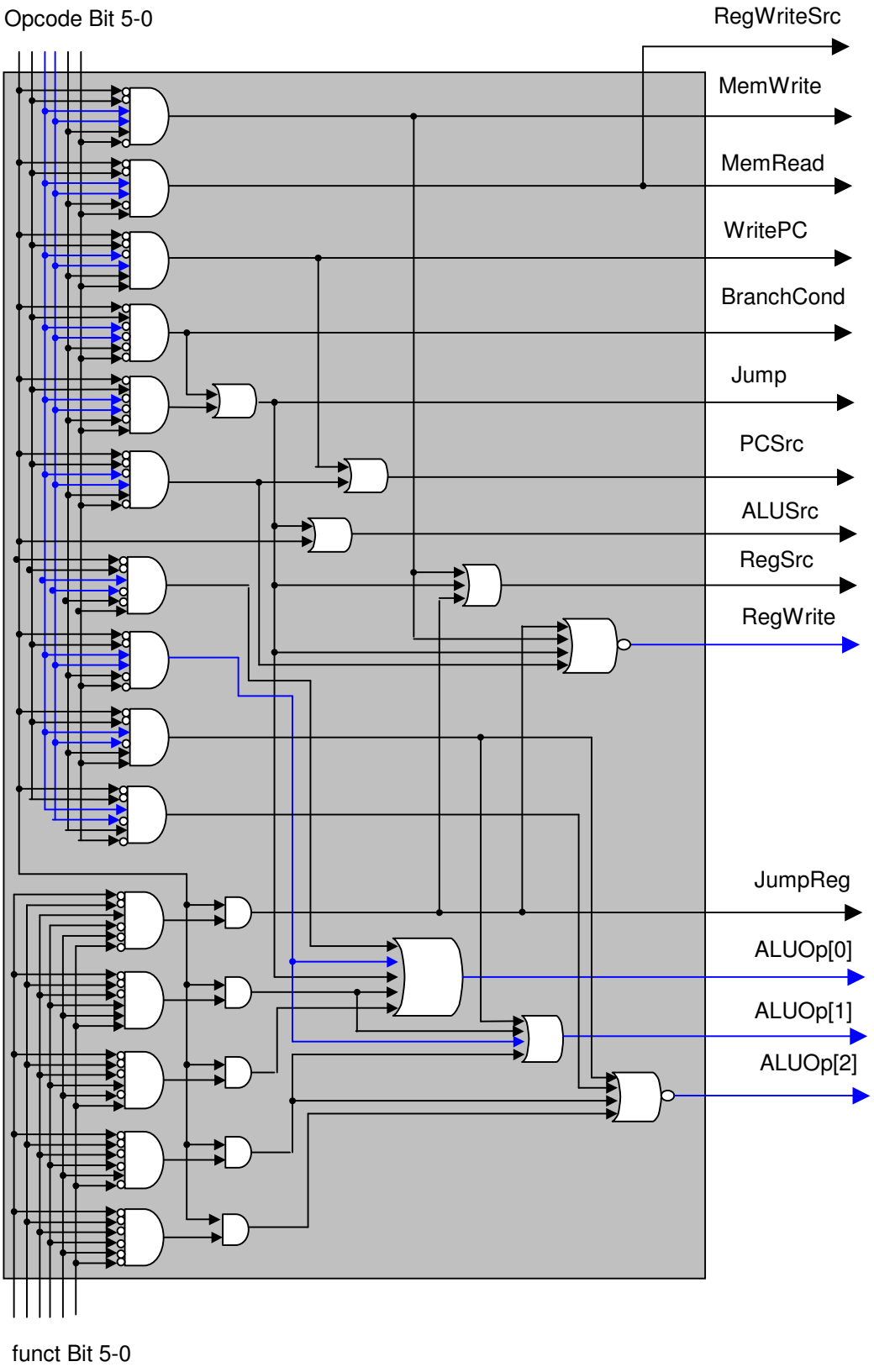
Befehl: andi
Opcode: 001010
funct: -



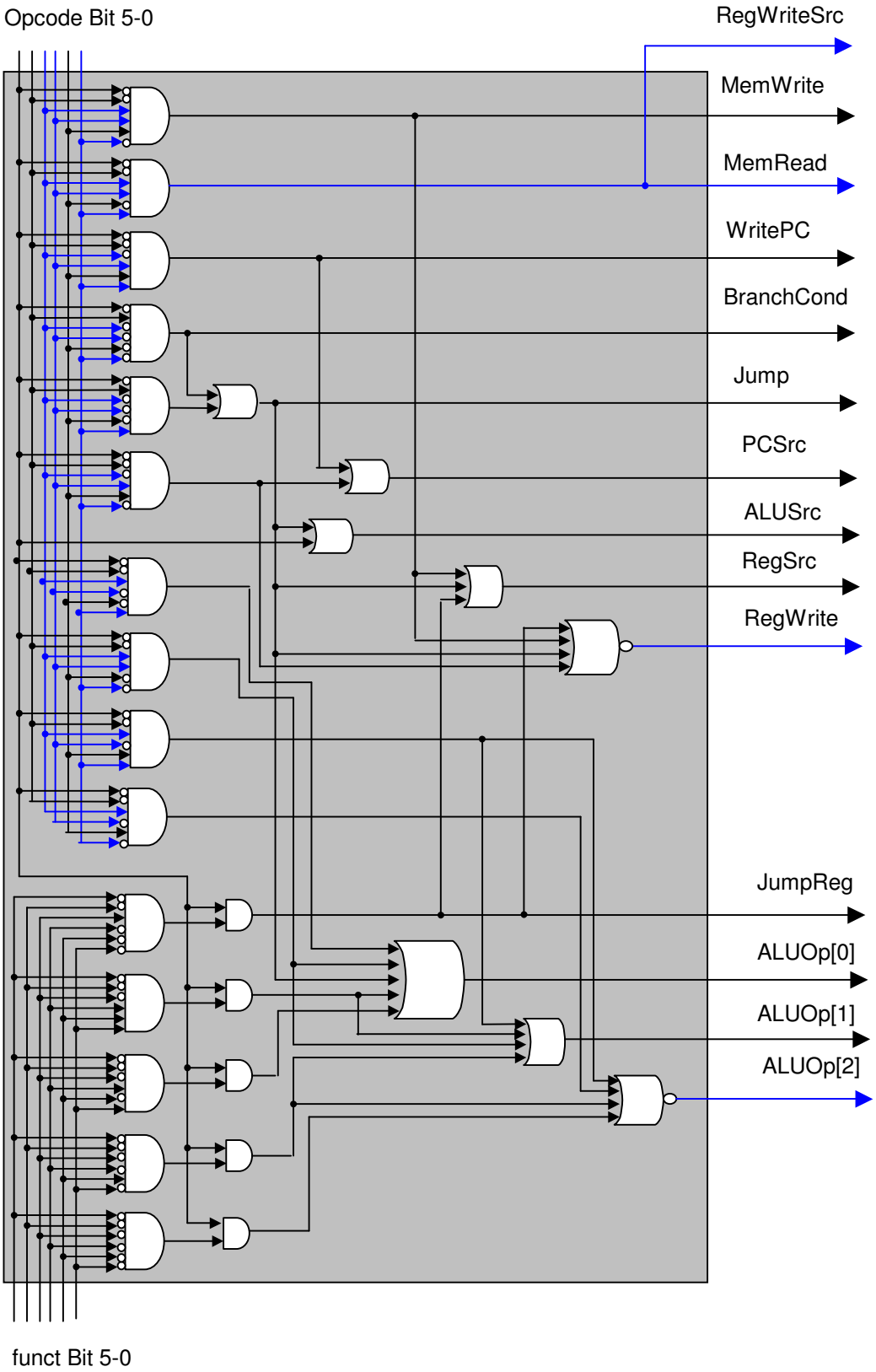
Befehl: ori
Opcode: 001011
funct: -



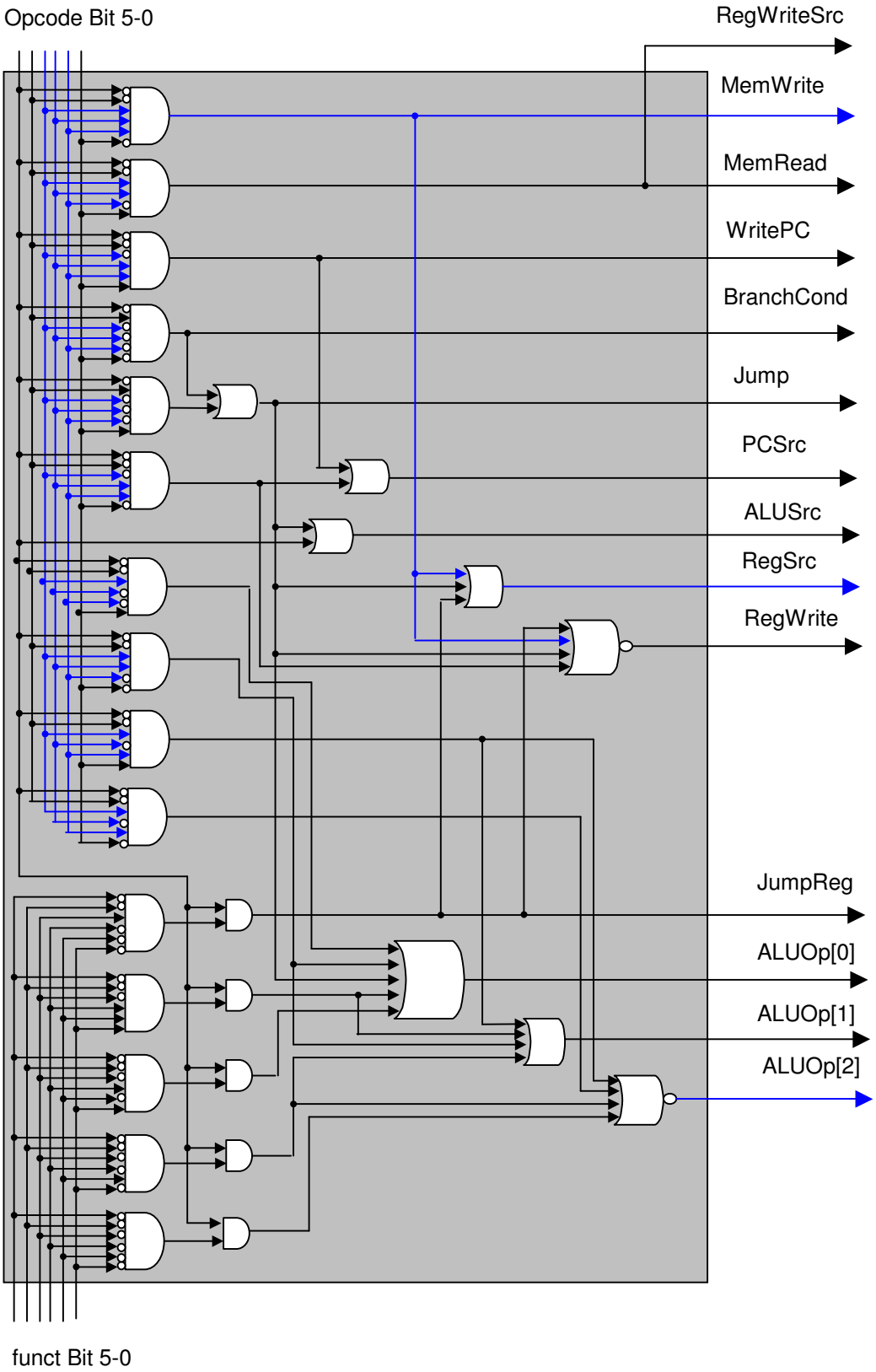
Befehl: slti
Opcode: 001100
funct: -



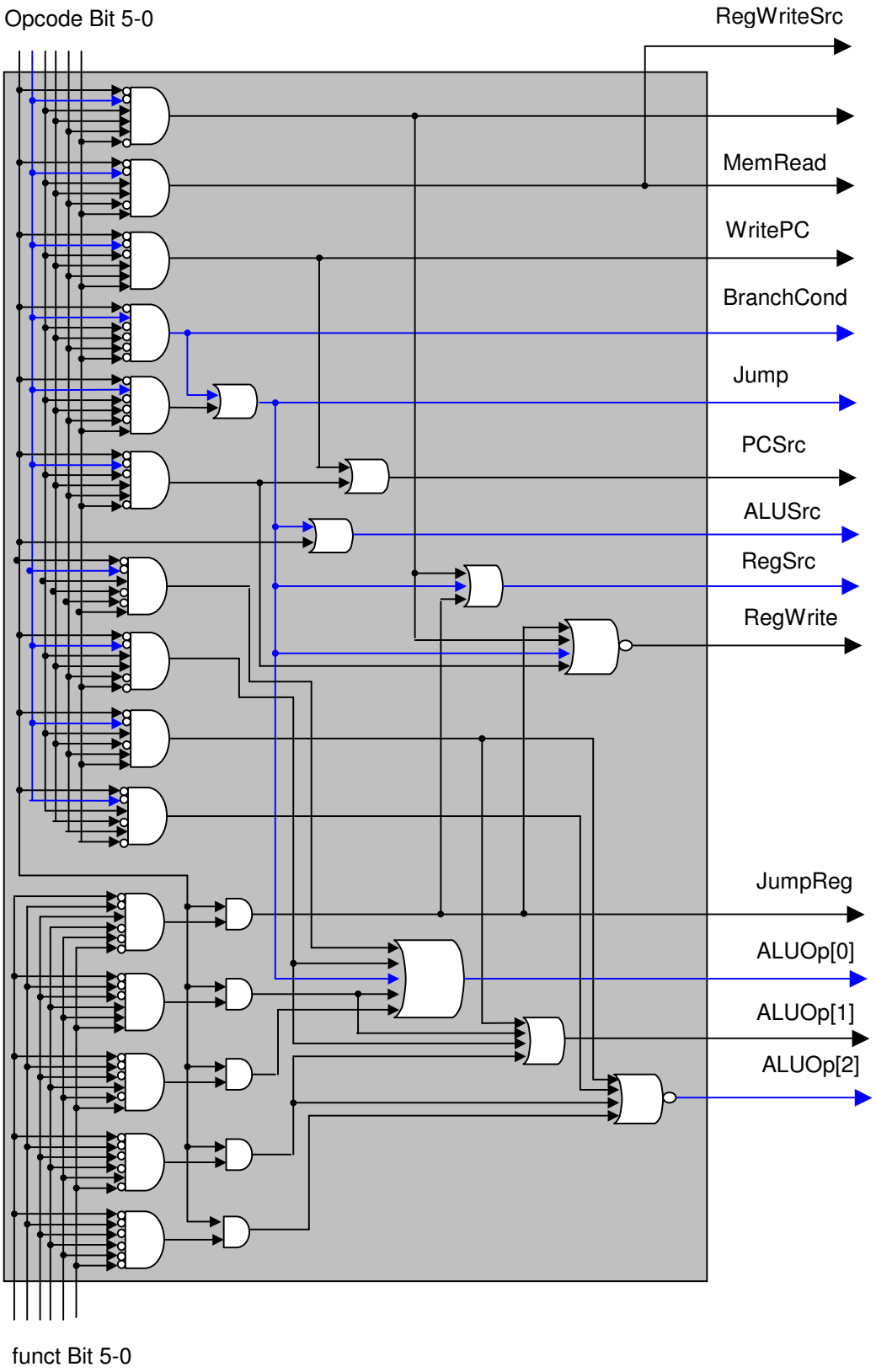
Befehl: lw
Opcode: 001101
funct: -



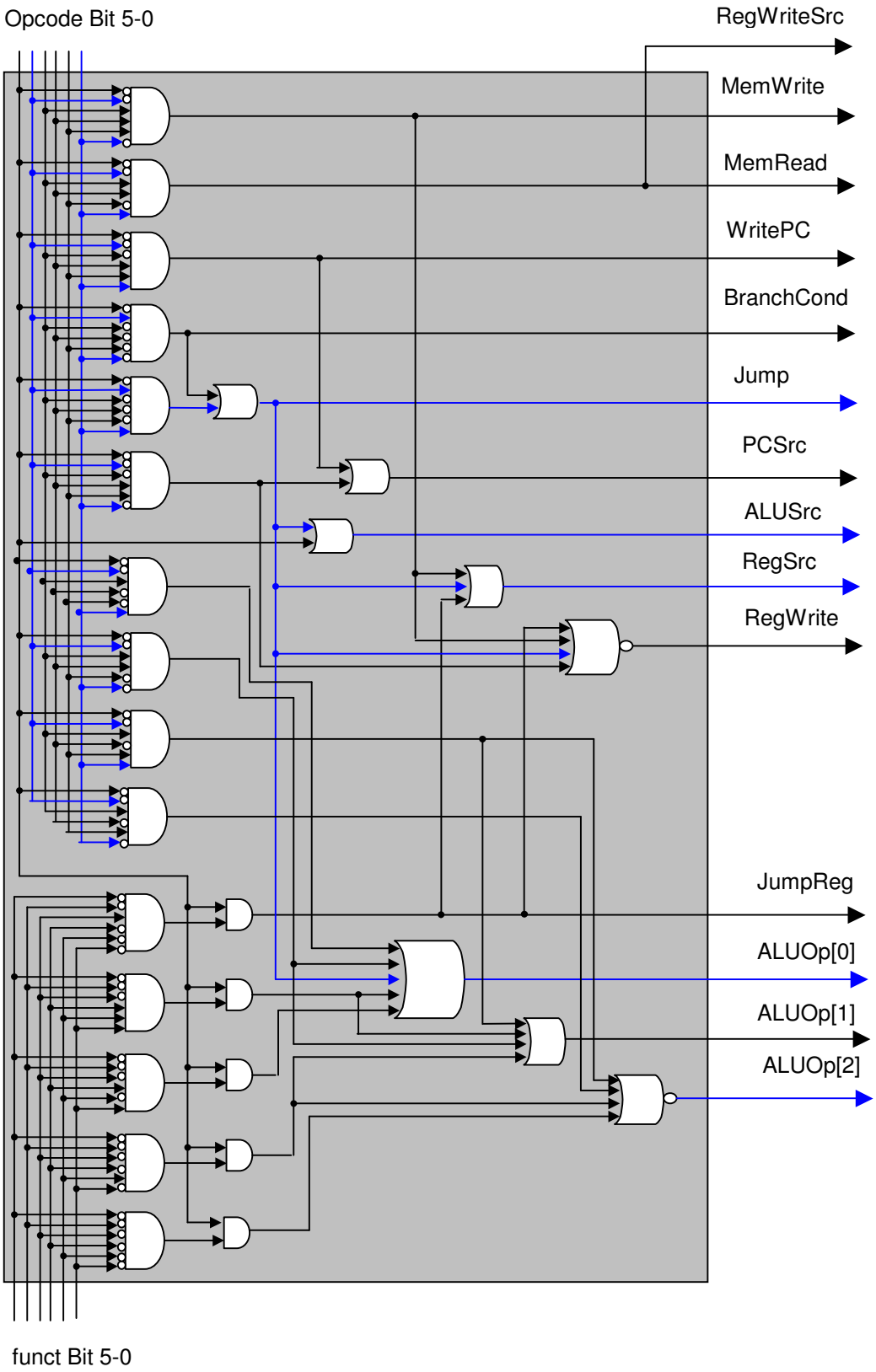
Befehl: sw
Opcode: 001110
funct: -



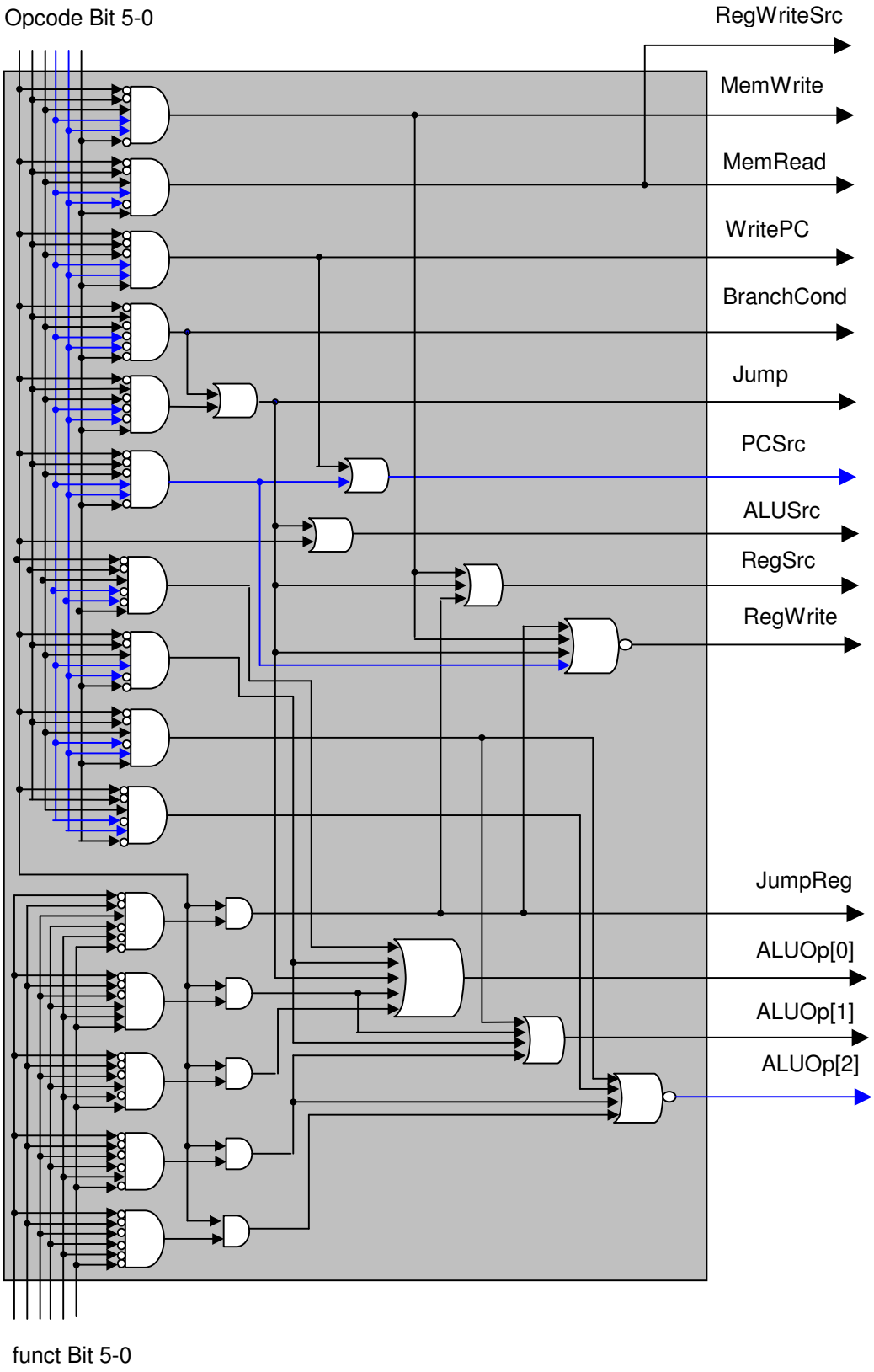
Befehl: beq
Opcode: 010000
funct: -



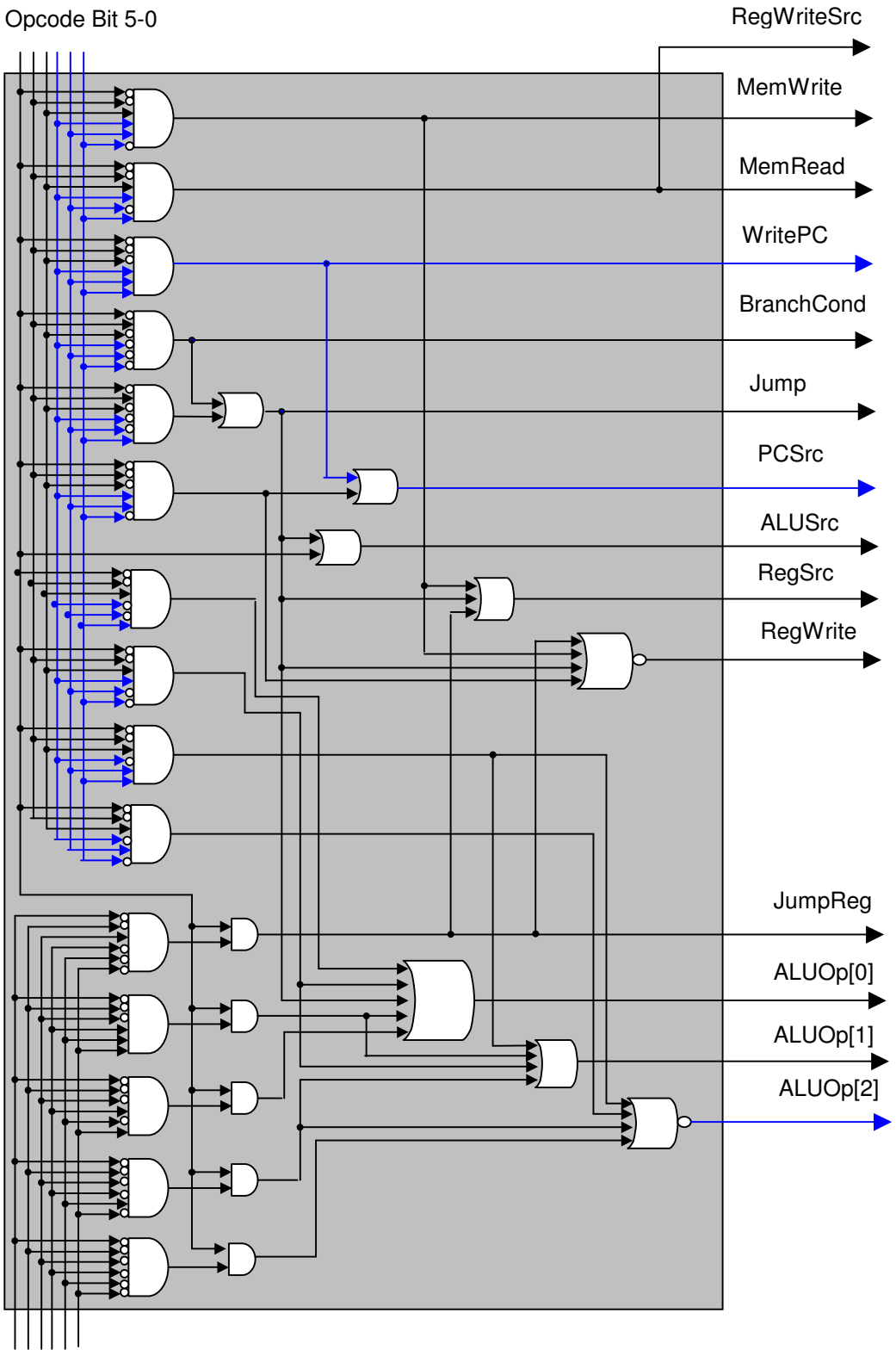
Befehl: bne
Opcode: 010001
funct: -



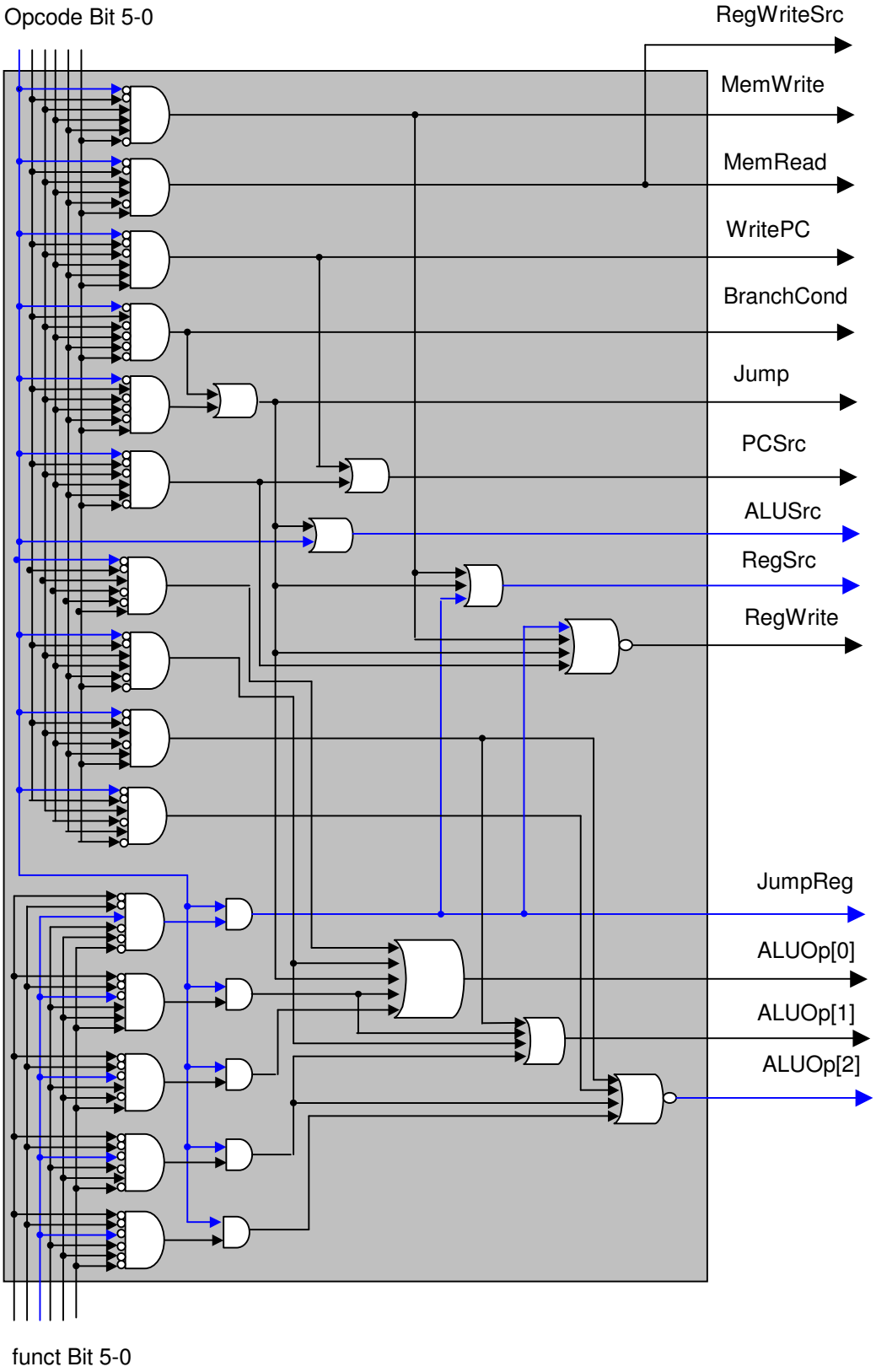
Befehl: j
Opcode: 000110
funct: -



Befehl: jal
Opcode: 000111
funct: -



Befehl: jr
Opcode: 100000
funct: 001000



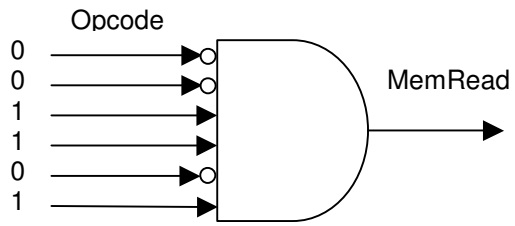
Aufgabe 5.1

- Das Kontrollsignal „**MemRead**“ wird nur zu einer 1 ausgewertet, wenn der Opcode den Wert 13: 001101 aufweist. Die Schaltung ist in a) gezeigt.

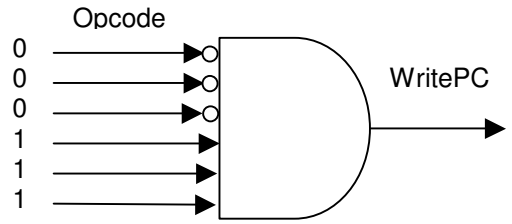
- Für eine 1 am Kontrollsignal „**WritePC**“ ist der Opcode 7: 000111 notwendig. Die Schaltung ist in b) gezeigt.

- Um eine 1 am Kontrollsignal „**JumpReg**“ zu erhalten, ist der Opcode 32: 100000 mit dem „funct“-Feld 8: 001000 Bedingung. Da der Opcode 32 der einzige Opcode ist, der in der Leitung mit Index [5] eine 1 trägt, genügt es hier, nur diese Leitung des Opcodes in die Schaltung einzubeziehen. Dafür müssen aber die Leitungen des Feldes „funct“ zusätzlich betrachtet werden. Die Schaltung ist in c) wiedergegeben.

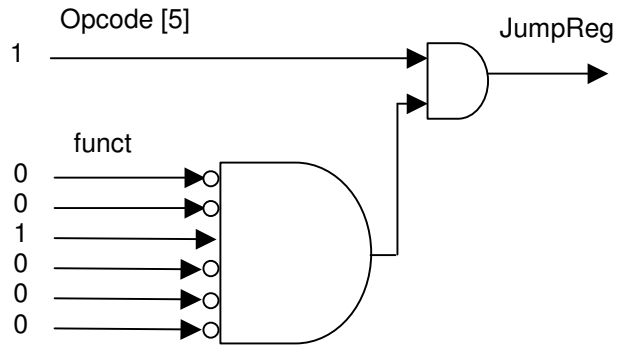
a)



b)

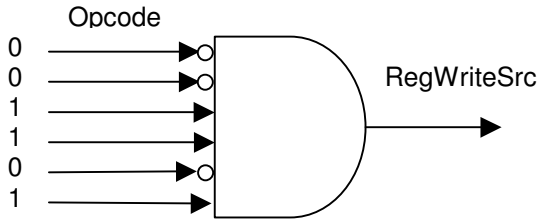


c)



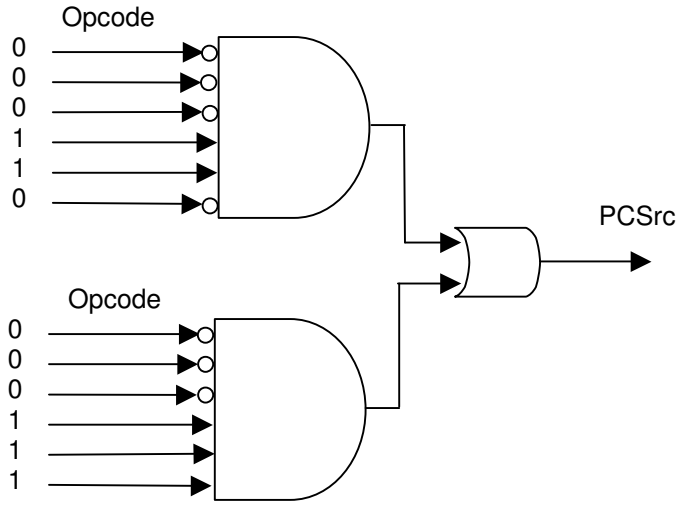
Aufgabe 5.2

Das Kontrollsignal „**RegWriteSrc**“ ist eine 1 falls der Opcode 13: 001101 angelegt ist. Die x-en der Spalte wurden mit 0 angenommen, dadurch wird die Schaltung für „RegWriteSrc“ identisch mit der Schaltung für das Kontrollsignal „MemRead“.



Aufgabe 5.3

Beim Kontrollsignal „**PCSrc**“ sind die benötigten Opcodewerte 6: 000110 und 7: 000111. Die Mehrweg-AND Gatter werden mit einem OR Gatter verknüpft.



Anhang C

Referenzen:

- Buch:
  David A. Patterson, John L. Hennessey
  Computer Organization & Design, The Hardware / Software Interface
  Morgan Kaufmann Publishers Inc., San Francisco, California 2005
  3rd edition, ISBN 1-55860-604-1

- Buch:
  David A. Patterson, John L. Hennessey
  Computer Architecture - A Quantitative Approach
  Morgan Kaufmann Publishers Inc., San Francisco, California 2003
  3rd edition, ISBN 1-55860-724-2

- Buch:
  Hans-Peter Messmer
  The Indispensable Pentium Book
  Addison-Wesley Publishing Co., 1995
  2nd edition, ISBN 0-201-87727-9

- Internet:
  www.mips.com
  Dokumentationen zu aktuellen und älteren Rechnerstrukturen. Eine Registrierung für die Einsicht in die Dokumentationen ist notwendig aber auch kostenlos.

- Internet:
  http://www.cs.wisc.edu/~larus/spim.html
  Hier gibt es einen SPIM-Simulator für die pipelined MIPS-Rechnerstruktur zu finden

- Internet:
  http://www.cpu-collection.de/?tn=1&l0=cl&l1=MIPS%20Rx000
  Übersicht über verschiedenste Rechnerstrukturen